Erstellung und Validierung eines Leitfadens zur Migration einer Microservice-Architektur nach Function as a Service

Frankfurt University of Applied Sciences

Gianni Pasqual

18.03.2020

Abstract

Mit der Technologie Function as a Service ist nach der Microservice-Architektur eine noch feinere Modularisierungsstufe von Software erreicht worden. Mit dieser ist es möglich Software noch schneller bereitzustellen und im gleichen Zuge Einsparungen bei dem Betrieb der IT-Infrastruktur zu erzeielen. Dies erscheint auf den ersten Blick natürlich sehr lokrativ für viele Unternehmen, jedoch sollte die Entscheidung für den teilweisen oder gesamten Umzug der Service-Landschaft wohl überlegt sein. Wie bei jeder anderen Technologie hat auch diese ihre Potentiale und Herausforderungen die gemeistert werden müssen. Für Unternehemn, die sich dazu entschieden haben Function as a Service (FaaS) in ihre Infrastruktur einzubinden, soll diese Arbeit als Orientierung bei der Migration dienen. Da viele Unternehmen bereits von den monolithischen Anwendungen auf kleinere modularere Services umgestiegen sind, geht diese Arbeit von einer bereits vorliegenden Microserive-Architektur aus. Es gilt herauszufinden in welchem Maße und ob überhaupt strukturelle Anpassungen vorgenommen werden müssen, als auch der Frage nach "Best-Practices", nach nun knapp vier Jahren des Bestehens, nachzugehen.

Zudem soll sich vor allem mit dem von den verschiedenen Anbietern ausgehenden Vendor-Lock-In auseinadner gesetzt werden und erarbeitet werden, welche Möglichkeiten bestehen, diesen zu umgehen bzw. zu mildern. Um mögliche Schwächen des Leitfadens aufzuzeigen, soll dieser beispielhaft an einem Service erprobt werden und auftretenden Fehler dokumentiert und behandelt werden.

Abkürzungsverzeichnis

ADF	Azure Durable Functions
ASF	Amazon Step Functions
AWS	Amazon Web Services
BaaS	Backend as a Service
BDD	Behaviour Driven Development
CaaS	Container as a Service
CD	Continuous Delivery
CI	Continuous Integration
CNCF	Cloud Native Computing Foundatino
DevOps	Development and Operations
DEV	Development
DSL	Domain Specific Language
FaaS	Function as a Service
IAM	Identity Access Management
IaaS	Infrastructure as a Service
INT	Integration (Testing)
IT	Information Technology
NIST	National Institute of Standards and Technology
OOP	Object Oriented Progamming
PaaS	Platform as a Service
PROD	Production
RPC	Remote Procedure Call
SAM	Serverless Application Model
SDK	Software Development Kit
UAT	User-Acceptance (Testing)
VCS	Version Controle System
XaaS	Anything as a Service

Inhaltsverzeichnis

1 Einleitung

1.1 Forschungsfrage und Ziele der Arbeit

1.2 Erläuterung der Problemstellung

1.3 Aufbau der Thesis

2 Aufarbeitung des Themengebietes

Ist eine Technologie, wie Function as a Service, noch relativ jung, so sind die Schritte der Findung einer in sich schlüssigen und allgemein vertretenen Definition oftmals noch nicht abgeschlossen. Auch bei der Definition von FaaS, Serverless und der Einordung dieser beiden Konzepte in die Infrastruktur des Cloud Computings8 [MG+11], ist dieser Prozess noch im Gange, wobei mitlerweile die unterschiedlichen Definitionen der offentlichen FaaS- und Serverless-Anbieter einige Gemeinsamkeiten aufweisen. Trotz alledem bestehen weiterhin Ungenauigkeiten, die es in den nächsten Jahren noch zu beseitigen gilt. Hierzu später mehr.

2.1 Definition des Begriffes Cloud Computing

Der Begriff Cloud Computing ist laut eines Technology Report bis auf das Jahr 1996 zurückzuführen, wo er in einem Business-Plan des Unternehmens Compaq von ein paar Entwicklern genutzt worden sein sollen, um über die zukünftigen Entwicklungen des Internet-Businesses zu diskutieren [Reg11].

Auch wenn das Konzept des Cloud-Computings, die dynamische, skalierbare, zuverlässige und unbegrenzte Bereitstellung von Ressourcen als Dienst über das Internet, immer mal wieder in der Literatur auftauchte [Fox+09], so erlangte es erst ab 2006 einen immer größter werdenden Bekanntheitsgrad. 2006 veröffentlichte AWS das Produkt Elastic Compute Cloud (EC2), gefolgt von Goolges App Engine 2008. Bereits in diesem früher Stadium wurde mit App Engine das Prinzip einer sog. „stateful“-Ebene, für das Speichern von des Aktuellen Anwendungsstatus und einer „stateless“-Ebene für die Ausführung des eigentlichen Programmes unterschieden [Fox+09]. Des Weiteren wurde 2009 in einer Berkeley-Studie die sechs größten Potentiale des Cloud-Computings herausgearbeitet welche bis heute in den unterscheidlichen XaaS-Konzepten umgesetzt wurden. Im folgenden sind diese kurz auf

gelistet.

1. Unbegrenzte Ressourcen wann immer sie benötgt werden.
2. Durch die Möglichkeit mit wenigen Ressourcen zu starten, vielen Nutzern zugang zu der Plattform zu gewähren.
3. Die genutzen Ressourcen so genau wie möglich nach dem „Pay Per Use“-Prinzip zu bezahlen.
4. Größenvorteile (Economies of scale) nutzen, um die Kosten durch eine dauerhafte, optimale Ausnutzung riesiger Datencenter auf ein Minimum zu reduzieren.
5. Operationelle Kosten so weit es geht zu senken und die Ressourcennutzung durch Virtualisierung so weit wie möglich auszunutzen.
6. Eine hohe Hardwareausnutzung durch „Multiplexing“ der Auslastungen verschiedener Unternehmen zu erreichen.

Das National Institute of Standards and Technology (NIST) veröffentlichte 2011, nach 16 vorausgehenden Definitionen, schließlich eine finale Definition des Cloud-Computings, welche bis heute als orientierung Anwendung findet und in einer vielzahl an Werken aufgegriffen wird [MG+11]. Die Definition nach NIST beschreibt das Cloud model als eines aus fünf essentiellen Charakteristiken und drei Service Modellen, IaaS, PaaS und SaaS bestehendes Modell, mit insgesamt vier Deployment-Models.

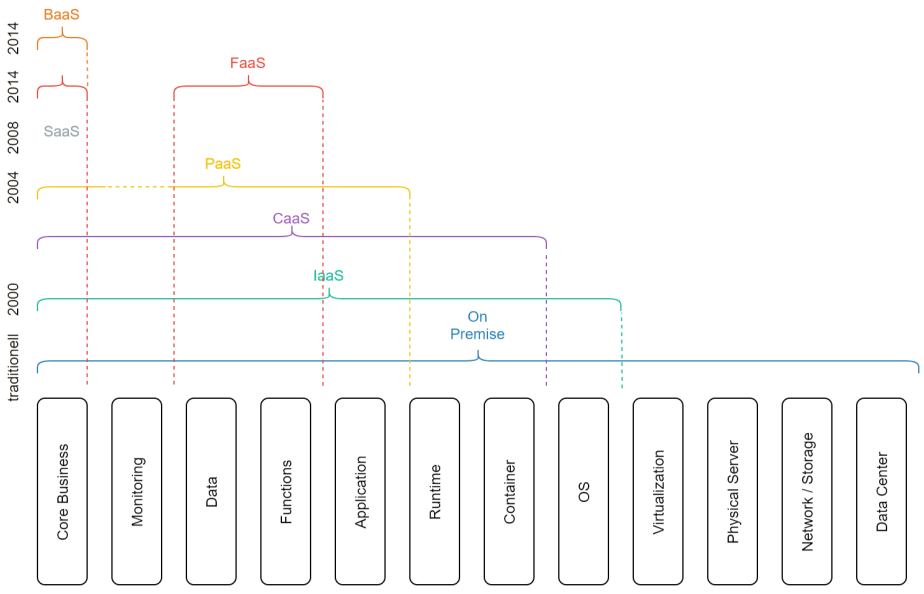
"Cloud Computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [MG+11].

NIST bricht die Definition mit Essentiellen Charakteristiken, Service-Models und Deployment-Models in drei Unterkategorien herunter, welche erfüllt sein müssen um dem Anspruch an Cloud Computing zu genügen.
Grob gefasst sind dies bei den Characteristiken, die automatische Bereitstellung der benötigten Ressourcen je nach bedarf (*On-demand self-service*), die Möglichkeit von allen gängigen Geräten darauf zugreifen zu können (*Broad network access*), die optimale Verteilung von Ressourcen auf die Kunden welche sie gerade benötigen, wobei hier die exakte lokation dieser von geringer

Relevanz ist (*Resource pooling*), die automatische Skalierung von Ressourcen (*Rapid elasticity*) und die Möglichkeit der Überwachung und Limitierung der Ressourcenausnutzung, welche für den Kunden als auch den Anbieter transparent sein muss (*Measured service*).

Bei den Service Models werden hier lediglich IaaS, PaaS und SaaS unterschieden. Laut NIST gibt es um den Code in der Could bereitzustellen vier verschiedene Deployment-Models, welche genutzt werden können. Mit der *Private cloud*, obliegt es dem Unternehmen seine Infrastruktur komplett selber zu betreiben, jedoch muss diese dem Unternehmen dabei nicht selber gehören, sondern kann von eine dritten Partei gehostet werden. Mit dem deployment auf eine *Community cloud* teilen sich verschiedene Unternehemen eine Cloud, welche entweder von ihnen gemeinsm, oder von einem Drittanbieter betrieben wird. Zuletzt gibt es noch die Möglichkeit die von einem Cloud-Computing-Anbieter bereitgestellte öffentliche Cloud *Public cloud* zu nutzen, bei welcher das Unternhemen die Räumlichkeiten bzw. Ressourcen von einem Dittanbieter nutzt. Das letzt Deployment-Model stellt die sog. *Hybird cloud* dar, welche eine Kombination aus den vorherigen darstellt [MG+11].

Abbildung 1: Übersicht Service-Models Cloud-Computing



Neben den bereits genannten Service Models IaaS, PaaS und SaaS gibt es jedoch noch weitere, welche sich im Laufe der Zeit etabliert haben. Abbildung 1 gibt einen Überblick hierüber.

2.2 Function as a Service

Function as a Service (FaaS) ist ein sogenanntes „Serverless“ Cloud-Computing Konzept, welches erstmals 2014 von AWS mit Lambda Functions als Preview Release und schließlich 2015 zur kommerziellen Nutzung zur Verfügung gestellt wurde [1]. Es kann nach IaaS und PaaS als ein weiterer Schritt in der Entwicklung des Cloud Computings gesehen werden, welcher das Management von Infrastruktur, Servern und Ressourcen weg von dem Entwickler nimmt und hin zu dem Cloud-Anbieter delegiert. Ein knappes Jahr später, 2016, traten Microsoft mit Azure Functions, Google mit Google Cloud Functions und IBM mit OpenWhisk in den bis dahin von Amazon dominierten Markt ein. Mittlerweile gibt es eine vielzahl an Open-Source sowie proprietären Anbietern, welche um die Gunst der Kunden werben und einen stetigen Wettbewerb aufrecht erhalten. Ein ausführliche Übersicht über die jeweiligen Anbiter findet sich online von der CNCF [2]

Da es sich mit FaaS und Serverless um eine noch recht junge Technologie handelt gibt es zu diesem Konzept keine offiziell dokumentierte Definition. bsp. von NIST o.ä., in der Literatur, wie es beispielsweise beim Cloud-Computing [MG+11] der Fall ist. Die Anbieter sind sich jedoch über die Funkitionalität von FaaS größtenteils einig, was das Spektrum an Funktionalitäten anbegeht. So definiert Microsoft Azure Functions als: „[...] event-driven serverless compute platform that can also solve complex orchestration problems. Build and debug locally without additional setup, deploy and operate at scale in the cloud, and integrate services using triggers and bindings. “ [3]. Hiermit beschreibt der Anbieter bereits die Kernfunktionalitäten von FaaS und drückt den Kerngedanken hinder FaaS aus. Das Konzept soll genutzt werden können, um bei dem Auftreten eines zuvor definierten sog. Triggers auf ein Event aus der eigenen Applikation oder der des Abieters reagieren zu können. Dabei stellen die meisten Anbieter wie Google, Azure oder AWS neben einem einfachen event-basierten HTTP-Trigger oder widerkehrenden zeitbasierten Triggern noch weitere, an die jeweilige Infrasturktur des Anbieters angepasste, Trigger zur Verfügung. Diese ergeben sich meist

[1] https://docs.aws.amazon.com/lambda/latest/dg/lambda-releases.html
[2] https://github.com/cncf/wg-serverless
[3] https://azure.microsoft.com/en-us/services/functions/

aus den BaaS Angeboten, hierzu mehr in *Abgrenzung von FaaS zu Serverless*, welche in Form von Datenbank Triggern bei einer CRUD-Operation, dem Anlegen eines Nutzers oder der Integration mit einem Push-Notification Service [AWS SSN, Google Pub/Sub etc.] auftreten können.

Abbildung 2: [4]

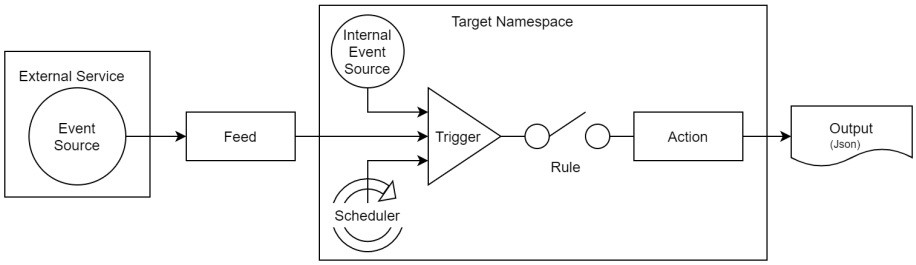


Abbildung 2 gibt hierbei eine Übersicht über die verscheidenen Event-Trigger, welche derzeit bei den großen, proprietären Anbietern (AWS, Google, Microsoft) existieren. Da hinter OpenWhisk mit IBM eine großes Unternehmen steht, wird dieses Modell in der Literatur häufig als Referenz in Bezug auf die anderen Anbieter verwendet. Es kann angenommen werden, dass andere Abieter ein ähnliches Konzept bei ihrer FaaS Architektur verfolgen, da Apache OpenWhisk in einer großen Cloud (IBM) bereitgestellt wird [VE+19]. Eine weitere fundamentale Eigenschaft ist die erwähnte Skalierbarkeit, welche nach dem sog. „Pay Per Use"-Modell abgerechnet wird. Gemeint ist hiermit, dass der Kunde nur das bezahlt was er auch wirklich verbraucht hat, wobei die Verrechnung extrem granular nach genutzten Ressourcen und gelaufener Zeit erfolgt. Der Kunde bezahlt daher nicht für die benötgten Ressourcen, wie er es bei PaaS (wobei es hier variierende und bereits granularere Modell gibt) oder IaaS der Fall ist, sondern nur für die tatsächlich genutzten Ressourcen.

Grundsätzlich ist zwar der Preis der für das Laufen einer einminütigen Funktionsaussführung verglichen mit dem Laufen des selben Codes auf einem von den Ressourcen her ähnlich bestückten PaaS-Server günstiger [Jon+19], jedoch der Anwendungszweck ein ganz anderer. Während mit PaaS Anwendungen bedient werden, die eine dauerhaft hohe Nachfrage erfahren, sollen mit Funktionen primär einfache stark frequentierende Aufgaben geläst werden, bei denen der Nutzer nur für die Ressroucen bezahlen muss die die

Funktion bei dem Aufruf nutzt und nicht für jene die ohen einkommende Aufrufe provisioniert wurden.

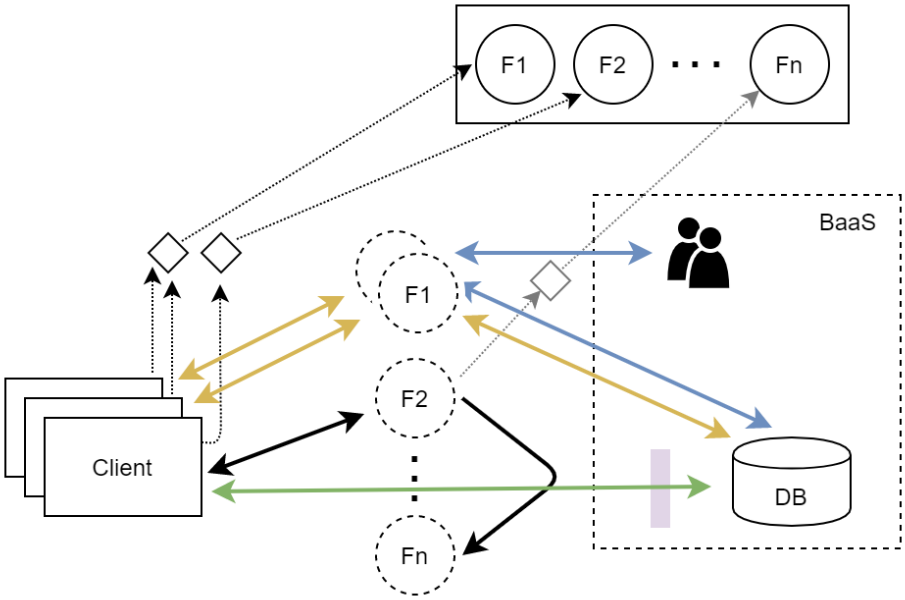
Ein weiterer wichtiger Punkt, welcher sich auch bei anderen Anbietern findet, ist in der Definition von OpenWhisk zum einen mit „[...] you can focus on building amazing and efficient applications [...]“ und zum anderen mit „developers write functional logic [...] in any supported programming language, that can be dynamically scheduled [...]“[5]. Ersteres meint die Abstraktion der Infrastruktur, welche dem Nutzer zum Bereitstellen von lauffähigem, skalierbaren, sicheren Backend-Code nicht bekannt sein muss. Die automatische Skalierung und optimale Verteilung von Ressourcen obliegt der Hoheit des Anbiters, und ist dem Nuzter nicht zugänglich, zumindest bei den proprietäre Anbietern. Letzteres ist zwar keine einzigartige Eigenschaft von FaaS, da dieses Konzept beriets lange bekannt ist und auch bei der Microservice-Enticklung häufig zum Einsatz kommt, jedoch ist die Auswahl aus vielen verschiedenen Programmiersprachen wie Java, Go, Javascript (Typescript), Python, PHP usw. (abhängig von dem jeweiligen Anbieter) unter dem Gesichtspunkt der direkten Nutzung zu sehen. Es muss keinerlei zusätzliches Setup oder andere Konfigurationen in der Umgebung vorgenommen werden, da dies der Anbieter übernimmt. Er kümmert sich um Patches und das Upgraden auf die aktuellste Version, was dem Entwickler ein großes Maß anflexibilität bei der Entwicklung der Anwendung einräumt. Eine weitere häufig in der Literatur referenzierte Defninition ist [Fow18], von Mike Roberts, welche die Beschreibung von AWS Lambda näher erläutert und im groben die oben genannten Punkte wiedergibt. Natürlich sind diese Eigenschaften mit einem gewissen Venodr Lock-In dieser jedoch weder als schlecht noch gut zu bewerten ist und vielmehr nüchtern auf seine Vor- und Nachteile anasystiert werden sollte. verbunden, da dieser Enstcheidungen über die Infrastruktur trifft, mehr dazu in *Potentiale und Herausforderungen*.

Abbildung 3 verdeutlicht hierbei noch einmal die Wirkungsweise von FaaS. Oben rechts ist der Funktionspool zu sehen, welcher bei dem jeweiligen Cloud Provider steht und in welchen der Entwickler seine Funktionne lädt. Wird eine Funktion von einem Client aufgerufen, so wird bei Aktivierung eines Triggers eine Kopie der jeweiligen Funktion instantiiert und alle mit diese Funktion verbundenen Abhänigkeiten geladen (bspw. benötigte Module). Je nachdem wie frequentiert die Funktionen von den Clients angefordert

[5] http://openwhisk.apache.org/

werden, werden die vorhandenen Instanzen automatisch von dem Provider

Abbildung 3: Anwendung mit FaaS und (P)BaaS, angelehnt an [SKM20]



hochskaliert, um den eingehenden Anfragen gerecht zu werden. Hierbei kann die Funktion eine Aufgabe direkt ausführen und das Resultat an den Client zurückgeben, oder wie bei F1 auf die im hintergrund laufende BaaS Infrasturktur zugreifen. Diese reicht von einfachen Datenbankabfragen bis hin zu anlegen eines neuen Nutzers etc. Limitiert werden die Möglichkeiten hierbei lediglich von dem Service-Ökosystem des jeweiligen Anbieters. Genauso ist es natürlich möglich Funktionen sich gegenseitig aufrufen zu lassen oder direkt mit BaaS-Servicen zu interagieren, siehe *Abgrenzung von FaaS zu Serverless.*

Um FaaS besser und einheitlicher definieren zu könne und damit einen plattformübergreifenden Standart zu schaffen, wird in der Industrie und Forschung das Verlangen nach einer Referenzarchitektur, wie sie beispielsweise für das Cloud oder Grid Computing vorhanden ist [Liu+11], [FK03] lauter. Die Abwesenheit einer solchen Architektur behindere die Etablierung von Best-Practices, Design Pattern und einen genaueren Überblick darüber zu erhalten, wie sich das Feld rund um Function as a Service entwickele

[Lei+19]. Erste Vorschläge wie die Referenzarchitektur von FaaS aussehen könnte, haben sich mit [VE+19] durch die über die Jahre, von 2016 an, gestiegenen Popularität von Faas in der Forschung und der Literatur bereits herausgebildet. War hier die Thematik bis 2016 nicht addressiert worden, so stieg die Präsenz von hier an kontinuierlich bis 2019, in Journals, Konferenzen und Workshops an [Yus+19].

2.3 Abgrenzung von FaaS zu Serverless

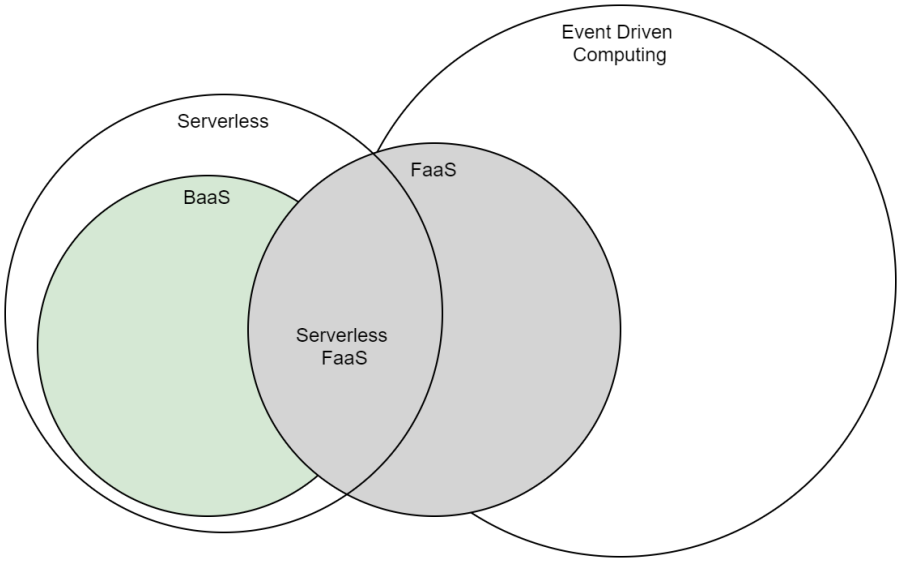
Serverless-Computing, oftmals auch als Serverless bezeichnet, ist eine Teildisziplin des Cloud-Computings, welche sich aus der Virtualisierung von Rechenleistung, Speicher und Netzwerken aus entwickelt hat [JC18]. Wie so häufig ist die Abgrenzung bzw. die Unterscheidung bei sich neu entwickelden Technologien nicht ganz einfach. Zunächst stand Serverless für Anwendungen welche teilweise oder komplett auf Drittanbieter zurückgriffen, auf sog. Backends as a Service (BaaS) [siehe Abbildung. 4], um serverseitige Aufgaben wie Datenbankabfragen, Nutzerverwaltung o.ä. zu regeln [Fow18].

Mit FaaS wurde das Serverless-Konzept dahingehend erweitert, dass die Serverlogik nicht mehr vollständig von einer dritten Partei zur Verfügung gestellt wird, sondern vom Entwickler selber implementiert werden kann. Serverless ist dabei eines der in den letzten Jahrne häufig genutzten Buzzwords in der IT, wobei der Begriff an sich etwas anderes suggeriert, als das was damit tatsächlich gemeint ist. Der Begriff impliziert die Abwesenheit von Servern, wobei damit lediglich eine Verschiebung der Zuständigkeiten einher geht. Entwickler einer „Serverlosen"-Anwendung bspw. mit FaaS, müssen sich nicht mit den operationellen Tätigkeiten wie dem *Provisioning*, Monitoring, der Wartung der Infrastruktur, der Skalierbarkeit dieser und der Robustheit des Systems befassen [Bal+17]. Jedoch geht mit der Abgabe an Zuständigkeiten auch ein gewisser Vendor Lock-In einher, wobei die Anbeiter darauf achten, das mit FaaS eine möglichst große Zahl der in ihrem Service-Ökosystem vorhandene Services genutzt werden [KS18]. Es sollen möglichst viele der bereits im vorherigen Abschnitt vorgestellten Trigger bei dem Bau von Anwendungen genutzt werden.

Function as a Service ist somit zum Großteil aus dem Event-Driven Computing, welches vor allem bei der UI-Entwicklung genutzt wird, abgeleitetes Konzept, welches Serverless-Computing adaptiert hat. Nichts desto trotz

wird der Begriff Serverless in vielen Fällen, laut [Lei+19] in 58% der Fälle, mit FaaS gleichgesetzt, was eine Abgrenzung erschwert. Abbildung 4 verdeutlicht in welcher Beziehung die unterschiedlichen Konzepte stehen.

Abbildung 4: Serverless Concept, including FaaS and BaaS



Besteht eien Anwendung in der Folge lediglich aus den zwei „serverlosen“ Komponenten, FaaS und BaaS, so ist häufig die Rede von einer „pure serverless“, also einer Anwendung, deren operationeller Teil vollständig an einen Cloud-Anbieter ausgelagert wurden und einer „hybrid serverless“ Anwendung unterschieden [Lei+19]. Bei letzterem fungieren die Kunktionen häufig als sog. „Glue“, der meisten zeitlich variierend Aufgaben übernimmt, da IDLE-Zeit, wie man sie aus IaaS oder PaaS kennt, nicht berechnet wird.

Nun mag der ein oder andere argumentiern, das ein Großteil der infrasturkurellen Aufgaben auch bei PaaS von dem Cloud-Anbieter übernommen werden, womit er natürlich recht hat. FaaS geht an dieser Stelle jedoch noch einen Schritt weiter. Bei Paas werden vorgefertigte *Packages* einer Anwendung auf der Runtime der entsprechenden Plattfrom bereitgestellt, womit sich die Entwickler immer noch im die Anwenungsstruktur kümmern müssen [Kap19]. In FaaS hingegen wird dieser Teil übernommen und der Entwickler konzetriert sich lediglich auf die Business logik und die jeweiligen Event-

Trigger.

Zuletzt soll noch auf einen weiteren Punkt, „NoOps“, eingegangen werden, der oft mit Serverless gleichgestellt wird. Es wird zwar durch die Übernahme des Loadbalancing, der automatischen Skalierung, Sicherheitsaspekten und Patches ein Großteil der Wartung ausgelagert, jedoch ist dies nicht mit der Annahme des Wegfalls der Operations von DevOps gleichzusetzten, wie suggeriert wird [Fow18]. Es obliegt weiterhin den Entwicklern qualitativ hochwertigen Code zu schrieben und diesen in der Cloud-Umgebung zu testen, um dessen Performanz sicherzustellen. Die Komplexität entfällt daher nicht vollkommen, sondern verlagert sich zu einem gewiessen Teil [Eiv17]. Schaut man sich beispielsweise AWS an, so muss eine vielzahl an Konfigurationen vorgenommen werden. Zwar kann später eine vorhanden auf alle Funktionen angewendet werden, jedoch ist diese in der Folge von essentieller Bedeutung in einer riesigen „Multitennacy“-Umgebung.

2.4 Potentiale und Herausforderungen

Die im Folgenden aufgelisten Potential und Herausforderungen, denen sich Literatur und Anwender von FaaS gegenübersehen, sind zu Teilen dem Konzept selber, zu Teilen aber auch der Implementation der jeweiligen Anbeiter bzw. Open-Source Lösungen geschuldet. Es werden daher auf Seiten der Herausforderungen diese zunächst erläutert und falls in der Literatur bereits adressiert, vorgeschlagen Lösungen aufgezählt. Bei beiden, Potentiale und Herausforderungen, wird sowol Bezug auf FaaS als auch auf BaaS genommen, da diese beieden Kategorien in den meisten Fällen in Kombination genutzt werden bzw. architektionisch beding in Kombination genutz werden müssen [siehe Abbildung 4 und „statelessness“ Abschnitt *Function as a Service*].

Potentiale

Kosteneffizienz

Time to Market / Lead Time Development

DevOps

Testing

Optimale Auslastung von Rechenzentren Greener Computing [SKM20] [Fow18]

Herausforderungen

Vendor Lock-In Auch wenn diese Eigenschaft sowohl **Physische Lokation**
Eine weiterer Punkt ist die physische Lokation der Funktionen. Dadurch, dass es dem Provider obliegt, die Ressourcen seiner Infrastruktur optimal zu nutzen, entscheidet dieser auch auf welcher Node eine Funktion ausgeführt wird. Platziert der Provider in der Folge Funktionen, die eine hohe Datenabhänigkeit haben, physische weit voneinander entfernt, so wird sich dies auf die Performance der Anwendung auswirken und letztlich als Latenz bei dem Endverbraucher zu spüren sein [SKM20]. Es gibt zwar bereits bei vielen Anbietern, wie AWS, Microsoft oder Google die Möglichkeit die Region beziehungsweise ein sog. Cluster festzulegen, jedoch lässt sich damit die Lokation lediglich eingrenzen, aber nicht rechenzentren-genau festlegen. Mit der Performance von „serverful"-Applikationen kann damit nicht gleichgezogen werden [SKM20].

„**Serverless**"
Es ist an dieser Stelle zwar relativ trivial, trotz alledem ein Nachteil verglichen mit Lösungen wie PaaS oder IaaS. Die Rede ist von dem Verlust der serverseitigen Optimierung. Wie bereits durch den Namen deutlich gemacht, existieren die Server für den Nutzer nicht bzw. hat er bei einer auf BaaS basierenden Softwarelösung keine Kontrolle über das Backend. Er kann lediglich die Datenbanken, Objektspeicher oder Authentifizierung an die Struktur seiner Daten anpassen und über „Regeln"Zugriffsbeschränkugen umsetzen. Die Services sind von dem jeweiligen Vendor vorgegeben und können bspw. nicht nach dem „Backend For Frontend"-Muster [6] auf die entprechenden Client-Typen, wie Tablet, Mobile-Phone oder Desktop angepasst werden.

[6] https://samnewman.io/patterns/architectural/bff/

Es existieren von jeder Sorte (Datenbanken, Objektspeicher, Authentifizierung usw.) nur eine Version auf welche man beschränkt ist. Jegliche benutzerdefinierte Logik muss daher in den Client ausgelagert werden, da es nicht auf backendseitig implementiert werden kann [Fow18]. Mit FaaS kann dieser Effekt gemindert werden, indem ein leichtgewichtige Logik in form von Funktionen serverseitig umgesetzt wird. Hierbei erfolg die Interaktion mit den unterschielichen BaaS-Servicen über die vom Anbieter beschränkten Trigger [siehe Abschnitt *Function as a Service*].

„**Statelessness**“
Mit der „statelessness“, also der Zustandslosigkeit von Funktionen, geht eine weitere Problematik von FaaS einher. Der Kern von Anwendungen ist es Aufgaben zu lösen, wofür viele verschiedene Schritte von Nöten sind, welche sinnvoll miteinander verbunden werden müssen. So ist es auch bei dem funktionsbasierten Aufbau von Applikationen wichtig, dass diese untereinander kommunizieren können und den Programmstatus voneinander abfragen können, ohne das es zu Inkonsistenzen kommt. Bedingt durch die kurze Laufzeit der Funktionen ist es essentiell, dass auch der Austausch in entsprechender Geschwindigkeit erfolgt. Wie in einem Report von Berkeley [Jon+19] herausgearbeitet, erweist sich das schnelles und exaktes „State-Sharing“ jedoch immer noch als problematisch dar, betrachtet man die Geschwindigkeit von „Serverless“-Anwendungen im Vergleich mit „Serverful“-Anwendungen.

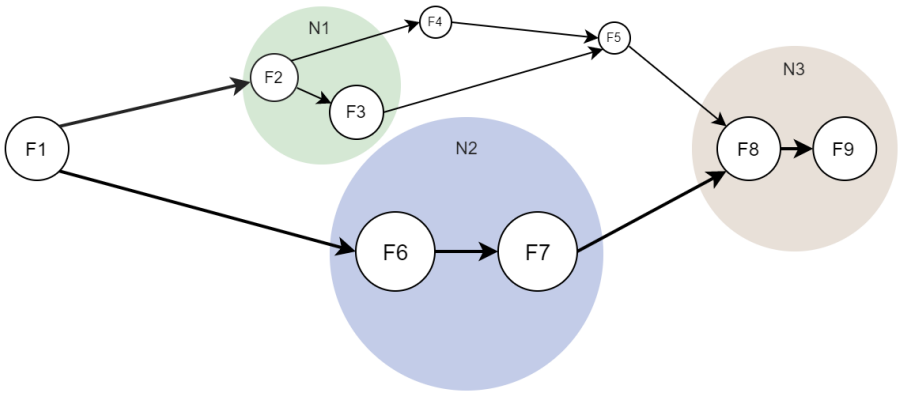
Grund hierfür sind die von den Anbietern zur Verfügung gestellten BaaS-Lösungen für das persistente Speichern des Programmstatus. Objektspeicher der verschiedenen Anbieter (AWS S3, Google Cloud Storage, Azure Blob Storage) sind zwar nicht teuter sehr teuter wenn es um das Speichern mehrerer GB geht, jedoch sind die Kosten beim Zugreifen auf die Speicher hoch und die Latenzzeiten von bis zu 20ms zu hoch [Jon+19]. Die Key-Value-Speicher der Anbieter sind in diesem Falle die bessere Wahl, da ihre Ansprechzeiten mit 8 - 15ms geringer sind, jedoch sind sie, bezogen auf ihre Input/Output Operationen pro Sekunde, deutlich teurer als die der Objektspeicher.

Neben dem schnellen Informaitonsaustausch der Funktionen mit einem externen Speichermedium, kann es bei einer Kopplung von Funktionen aus performancetechnischen Gründen sinnvoller sein, den aktuellen Programmstatus direkt and die nächst Funktion weiter zu geben. Wie zuvor bei der *physischen Lokation* der Funktionen erwähnt, ist es wahrscheinlich, dass

Funktionen nicht auf der selben Node gestartet werden. Der Loadbalancer des jeweiligen Providers entscheidet je nach Auslastung der Infrasturktur, auf welcher Node eine Kopie der Funktion ausgeführt wird. Auch wenn im Idealfall die Weitergabe des „application states“ schneller ist als der Zugriff auf ein externes Speichermedium, so wird dies durch „Startup-Latencies“und physischer Distanz wieder relativiert. Um dem entgegenzuwirken schlagen Shafiei et al. [SKM20] vor, mit einem *Dependency-Graphen* miteinander verwobene Funktionen zu kennzeichnen und direkt bei dem initialen Aufruf einer Funktion mögliche Folgefunktionen zu starten.

Würde man dem Nutzer die Möglichkeit geben zu bestimmen auf welcher Instanz (Node) eine Funktion laufen soll, so das Konzept von FaaS damit untergraben werden (siehe *Optimale Auslastung*). Es würde dazu führen, dass erneut Kapazität zurückgehalten wird, über welche der Provider nicht mehr frei verfügen könnte.

Abbildung 5: Dependency Graph nach [SKM20]



Wenn jeder Nuter bestimmen könnten auf welcher Node, bspw. Node-X, seine Funktionen laufen sollen, um den Progrmamstatus mit so wenig physisch bedingter Latenz wie möglich weiterzugeben, so wäre eine optimale Auslastung nicht mehr garantiert. Der Provider müsste stets Kapazität von Node-X zurückhalten und könnte sie nicht für andere Nutzerfunkitonen freigeben. Er muss stets einen Teil der Kapazität von Node-X zurückhalten, für den fall dass inaktive, aber der Node zugeteilte Funktionen aufgerufen werden. Mit einem *Dependency-Graphen* wie in Abbildung 5 zu sehen würde dieses Problem umgangen werden. Der *Load-Balancer* des Anbieters könnte

unbeinträchtigt die Funktionen wie gewohnt je nach Auslastung auf freie Nodes verteilen, da er die korrelierenden Funktionen „gleichzeitig“ starten kann. Da die Latenzen beim Starten eines Services erheblich höher sind, wobei die Programmiersprache dies zusätzlich beinflussen kann [Man+18], als jene durch die Lokation bedingt [Adi+19] [JC18], würde deren Eliminierung die Weitergabe des *States* erheblich beschleunigen.

Generell betrachtet könnte die Geschwindigkeit der Kommunikation von Funktionen untereinander beschleunigt werden. Verbindet man das ganze noch mit einem einfachen neuronalen Netz, um beispielsweise vorherzusagen mit welcher Wahrscheinlichkeit Funktion zwei (F2) im Vergleich zu Funktion sechs (F6) angesprochen wird, so könnte die Performance sogar noch ressourcenschonender auf Seiten des Cloud-Providers gestaltet werden. Zunächst käme es entweder zu einer höhreren Belastung, sollte man zu Beginne alle Funktionen starten oder zu einer höheren Latenz, sollte man die Funktionen einzeln starten und aus dem sich ergebenden Verlauf lernen. Nach einer kurzen Lernphase würde die Auslastung der Infrastruktur jedoch erhöht werden.

„**Multitenancy**“
Obgleich das Konzept von FaaS darauf beruht, dass der Anbieter eine Vielzahl von Funktionen verschiedener Kunden steuert und pflegt, so soll sich jeder Kunde fühlen, als sei der der Einzige. Allerdings stellt dies die Betreiber der Plattformen vor weitere Herausforderungen, da sie die Ressourcenisolation sowie Zugriffsbeschränkugen zuverlässig gewährleisten müssen. Dies ist jedoch leichter gesagt als getan, da es teilweise nicht nur von dem Provider selber, sondern auch von der richtigen Konfiguration der Sicherheitseinstellungen der Nutzer [7], [8] abhängt, ob die Daten unzugänglich für Dritte sind.

Obwohl Anbieter wie AWS Mittlerweile erfahren genug sein sollten, dass solche Probleme nicht mehr zu erwarten sind [Fow18], ist weiterhin Vorsicht geboten. Es nichts an der Tatsache, dass auch sie sich mit Sicherheitsaspekten, der Robustheit und Performance ihrer Infrastruktur ständig weiter auseinadner setzten müssen. Kein Kunde darf die Daten eines anderen sehen, kein Fehler die Stabilität anderer Funktionen gefährden und kein plötzlich auftretender *Spike* eines Kunden die Performance der Funktionen eines anderen beeinträchtigen. Das Zusammenspiel von RPCs und der Con-

[7] https://awsinsider.net/articles/2017/06/20/voter-data-leak.aspx
[8] https://searchsecurity.techtarget.com/news/450422962/Another-AWS-cloud-data-leakage-due-to-misconfiguration

tainersicherheit muss dauerhaft gegeben sein und durch sorgfältiges *Security Managament* sichergestellt werden [MB17].

Testing

Cold-Starts

Cold Starts beziehen sich auf das Starten eines Containers, in welchem eine Kopie der benötigten Funktionen zur Verfügung gestellt wird. Ist eine Funktion lange nicht genutzt worden oder wird zum ersten Mal angesprochen, so muss für die Ausführung zunächst alles vorberietet werden. Vor allem bei sehr kurzen Funktionen stellen sie ein erhebliches Problem dar, da ihre *Start-Up*-Zeit bis zu ein Zehnfaches der eigentlichen Ausführungszeit einnehmen kann [SBW19]. Ein Container muss auf einer Node gestartet werden und die Funktion, sowie deren Abhänigkeiten geladen werden. Die Zeit die es dauert, bis die Instanz bereit ist hängt dabei von verscheidenen Faktoren, wie der Programmiersprache, der benötigten Bibliotheken (*Libraries*), der Größe der Funktion (menge an Code) und der Hardware [SKM20] [Jon+19] ab, auf der die Funktion instantiiert wird. Wird eine Funktion dabei sehr frequentiert aufgerufen, so halten die Plattformen diese Funktionen für eine gewisse Zeit vor („warm“, bei AWS bis zu einer Stunde [RC17], wodurch die Wahrscheinlichkeit eines erneuten *Cold Starts* gegen Null tendiert. Bei Funktionen die hingegen nur einmal pro Stunde bemüht werden, sind die Folgen von *Cold Start* deutlich häufiger zu spüren [RC17].

Um dem entgegenzuwirken schlägt [Cas+19] vor, eine Art stammzellen Container vorzuhalten. Diese würden für die unterstützten Programmiersprachen bereits instantiiert aber leere Container vorhalten. Diese Container sind nicht Kundenspezifischen und generisch nutzbar. Damit würden zwei der obene genannten Punkte, die Programmiersprache und die Hardware, welche die *Startup*-Zeit der Funktionen negativ beeinflussen, eleminiert werden. Lediglich die größe der Funktionen und deren *Dependencies* müssen noch geladen werden. [IMS18] hat zudem gezeigt, dass mit der Erhöhung der Speicherkapazität die Zeit bis eine Funktion bereit ist einen *Request* zu bearbeiten verringert werden kann. Kombiniert man dies mit dem Stammzellen-Ansatz, so wird das Laden des Programmcodes und der *Libraries* beschleunigt. Natürlich kommt der zusätzliche Speicher nicht umsonst, jedoch liegt es in dem Fall bei dem Unternehemn zwischen den Mehrkosten und dem Performancezuwachs abzuwägen.

3 Guidline on Migration

The below-presented guideline for migrating parts of existing microservice architecture to the relatively new cloud-computing concept, Function as a Service, is structured as follows. At first, the decision to migrate is questioned by challenging the motives of adopting this new technology and giving advice on when to migrate. Afterward, criteria for selecting the right service will be stated, which gear towards the possibilities that come along with FaaS. When a reasonable service has been found, a provider needs to be identified, which best meets the company's requirements. There are two types of providers that primarily differ in the service-ecosystem inherent to them and the degree of vendor lock-in. Whereas proprietary platforms like AWS, Azure, or Google provide a vast ecosystem of feature services and backend as a service offering, open-source frameworks like OpenWhisk, Fission, or Kubeless dedicate greater control to the developers. When implementing FaaS, the open-source solution can be far more adjusted and configured to best suit an organization's needs. Next, impacts on the general process of software development, the „Dev"-part in *DevOps*, will be investigated. Changes in organizational structures regarding the development team, as well as effects on the CI/CD pipeline, challenges faced with local testing, and the VCS, will be addressed. With the development process covered, the guideline on migration will end with the operational part, which will be taken into account. Alternations on the process of monitoring, maintenance, and testing instances will be covered, and advice on coping with them will be given.

To better follow along with the guideline, the subsequent section will provide an overview of the current microservice infrastructure. The development process, as well as the operational tasks, will be described, along with the underlying service-landscape.

3.1 Zu grunde liegenden Architektur

Um im weiteren Verlauf dieser Arbeit Aussagen über die nötigen Anpassungen und Auswirkungen auf das Unternehemen, die Entwicklung (Dev) selber, sowie die Kultur und den operationellen Teil (Ops) treffen zu können, soll zunächst der momentan vorliegende Aufbau beschrieben werden. Hierbei handelt es sich einfach gesagt, um die Entwicklung von Microservices, wobei momentan 31 Services in unterschiedlichen Versionen und Testinstanzen vorliegen.

3.2 Challenging the Decision

Whereas migrating from a monolithic application to serverless, respectively, Function as a Service, seems far more challenging, due to the assumed size and complexity of the application, migrating from a microservice-architecture can be challenging as well. With a monolithic application, functionalities need to be identified in the first place and afterward be broken down into many small functional sections. Microservices, on the other hand, might already be small enough that they supply only certain functionalities. Still, in some cases, they do incorporate too many functionalities to be converted to functions right away. Nevertheless, dealing with the issue of outsourcing the currently inherent state of services, as well as reducing dependencies and optimizing code remains.

Before starting to introduce this new technology into the service landscape, it is essential to contrast the current state of the system with the desired state of the system. When a system is already composed of many small services, the underlying infrastructure likely consists of VMs or containers. Those containers often run in a PaaS environment, and it is not possible to scale them to zero to free reserved computing capacity in favor of other services to consume it. To do so or even to outsource the expense of provisioning services, managing the underlying infrastructure and coping with operational tasks, such as monitoring, load balancing, etc., Function as a Service can help to achieve this desired state. The restrictions that come along with the ease of development have to be traded off against its benefits.

To profit as much as possible of introducing FaaS, the services, which should be migrated, have to be inspected regarding dependencies, size, version, language, and complexity. It is recommended to reduce dependencies them to a minimum, due to their effect on the star-up time of the container the function runs in [Man+18]. Especially with Python, Nodejs, and Java [PS17], loading all dependencies required, which might again interconnect with other dependencies, the number of dependencies will have an impact on cold starts. The next issue which needs to be addressed is state management.

As mentioned in section two, *benefits and drawbacks*, there is no persistent state an application can rely on. There actually is a persistent state in the container of a function, but whether an incoming request will hit a certain, running container, is unpredictable. When a service does not receive any request or is running for a long time, the provider will eventually kill its

container to free capacity. In the case of AWS Lambda, the current maximum amount of time, of a frequently called function until it gets killed, is 45 minutes [9]. Later on, when comparing cloud providers with open-source frameworks, there will be guidance on fining an appropriate solution to meet a company's purpose. Still, for now, the following must be considered. When using an open-source framework like OpenWhisk, the external state management system can be covered with Redis or another low latency database. Whereas using a cloud provider platform, one of its integrated database solutions will presumably be the best choice. If it is necessary to have full control over the performance and configuration of the database, an open-source framework has to be chosen.

The latency and frequency of a service will also decide upon its aptitude for being a candidate to get migrated to FaaS. If the service is called very frequent and experiences most of the time very high traffic, the concept of Function as a Service will not apply to it [Jon+19]. Due to a limit of concurrent running functions, that can vary between the different providers and frameworks; incoming traffic can only be handled upon a certain amount. Moreover, the same application experiencing the same amount of traffic, once running on a FaaS platform and once running in a docker container in a PaaS environment, will be more expensive implemented with FaaS, than it will be with PaaS [Jon+19]. Therefore, services with various workloads, having eventually high peeks and then some time of inactivity, are more applicable to the concept of serverless, than their counterparts.

Latency should also not be a critical component due to the before mentioned cold starts. Latency sensitive applications like trading platforms, which strongly rely on real-time data, are not a suitable candidate for Function as a Service. Another pitfall is the promise of not having to maintain, provision, scale, or monitor the infrastructure and thereby reducing complexity and operational tasks. On one side, the cloud provider will, to some extent, take care of load balancing, scaling functions up and down, and providing monitoring solutions. On the other side, new complexity will appear in other areas. The issue of mono-repo and poly-repo arises, as discussed in a later section, followed by the necessity to learn the provider's DSL. Furthermore, the team needs to acquire skills to correctly configure security policies, interconnect function with each other, define different triggers on functions,

[9] https://aws.amazon.com/de/lambda/

deal with concurrency settings [10], and many more aspects add complexity to new fields. If the company aims at reducing infrastructural complexities, then Function as a Service can support at the listed tasks, but when its primary purpose is to reduce operational efforts, as often suggested with the term „NoOps“ [Eiv17], FaaS, again, is not the right choice.

Lastly, attention needs to be drawn to the programming language and lead time development. FaaS can provide a reduction in lead time development and time to market, thanks to its small codebase and rapid development [SS18], [Lei+19]. Developers can make changes, fix issues, test, and finally deploy a new version to production in a view minutes. Prerequisite, even though this might sound trivial, is the platform language support for the language used in the application or service. Although the big cloud providers already support many programming languages, application logic might have to be rewritten, if that specific language is not supported. In the case of version incompatibility, the application needs to be rewritten as well. Especially when the provider's version is below the version of the application, it becomes an issue.

3.3 Choosing a suitable service

The service for starting the migration should manifest specific characteristics. As mentioned above, and in section two, *benefits and drawbacks*, there are a view things that need to be taken into account. Due to Javas CPU-intense runtime and the time for loading potentially large amounts of libraries, the language is likely to raise the time of cold starts, eventually resulting in higher latencies [BRH18]. Also, the object-oriented model of Java must be restructured to meet the requirements of the original functional programming style of FaaS. Concepts embraced by Java, such as getters, setters, empty methods, constructors, and singletons, need to be considered when mapping an object-oriented programming language to decoupled functional units.

Moreover, migrating from an object-oriented approach, respectively Java, state changes in classes and objects need to be treated separately. In a functional unit, keywords such as *this*, referencing itself in a java-class, are not applicable. To address this issue, the self-reference, as suggested by Wright et al. [Wri+98], should be provided with the method signature. This way,

[10] https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html

the same underlying object structure can be equally invoked with every new instance of a function getting started. Operations on the container do not interfere with objects in other containers, and saving the state in external databases will not cause redundancy between the different objects [SD17]. Considering these additional steps of migrating a service, written in an object OOP language [Lei+19], staring with one written in a functional programming language might be easier. The development team has to decide on the issue of refactoring in favour of rewriting. When the goal is to maintain consistency in language, accross the entire platform, the former decision is in favour. Otherwise, converting the service into a functional language, by rewriting it, could be another option. In the latter case, JavaScript, PHP, and Python, which primarily are functional, are suitable candidates for conversion.

Furthermore, the majority of practitioners use functional programming languages over OOP languages [Lei+19], increasing the likelihood of finding solutions to problems on the internet. In contrast to legacy applications and the majority of PaaS and IaaS architectures, the service that is about to be converted should not experience consistently high loads. Services like the CI/CD pipeline, experiencing various workloads are suitable to be turned to FaaS. Besides that, looking at the existing microservice architecture and identifying services that spend most of their time in IDLE, can be considered for migration.

3.4 Effects on teams and development

When a microservice architecture is prevailent, FaaS can be adopted with less changes as if the existing infrasturcture is a monolithic architecture [Fox+17]. Concepts like agile development, continues delivery and continuous integration as well as differen mindset amongst the development team are are probably more common. Nevertheless, FaaS goes a step further, so will be changes to the PaaS-adapted developemnt. Starting off with agile development and „DevOps“, Function as a Service forces companies and teams to adapt an agile mindset and agile techniques [Ben+18]. The neccessity of iterative cycles is accounted for by the concepts gerater modularity, compared to microservices. Persuing prodcedures form older waterfall dirven prjects are still prevailent, would not satisfy the core-features of Serverless, respectively FaaS. With a significant reduce in lead time development, long periods of requirement engineering will hinder the effectiveness of FaaS. Teams teams which already incorporated agile process frameworks, such as

scrum, will have an advantage over those reamining to acquire experience. Serverless development will enhance the inportance of continuous delivery and continuous integration of integrating procedures like unit testing and integration testing, to diminish time to markted to a minimum. Depeding on the size and complexity, which sould be kept to a minimum as well, chances are that FaaS will be considerably faster than the current microservice development.

Furthermore, the size of the teams, to be more accurare, the number of developers working on a function, should experience a decrease. Whereas now two to three Developers are working on a microservice, separation of concerns, naturally imposed by FaaS, will futher reduce complexity. Also, by reducing complexity and focussing on a specific functionality can help to further optimize each function. By doing so, besides the reduce in startup latency, the shorter the execution time is, the less will be chared by the paltfrom [SKM20].

However, when maintinaing a microservice infrasturcture in most cases each serveice and each version of a service, is likely to have its own repository. With Function as a Service, this approach will be challenged regarding the number of fuctions, that make up the application or service. When having a repository for each service, odds are in favour of experiencing an overhead in version controle and complexity, when maintaining the landscape. Hence teams pursuing this approach are facing duplication in their code base, because transparency of already existing functions, can not be gathered with littel effort [Rac+19]. This seems to be the cause why many users follow a mono-repository approach, over having many small repositories [Bro19].

With that said, FaaS seems to be conflicting with itself to some extend. Altough ist modularity suits the *poly-repo*-approach quite well, the accompanying complexity, when an application keeps growing, can cause redundancy and inefficiency. As a suggestion, ther should be a mono-repository-approach for each (scrum)-team and a ploy-repo approach for each division in a company, coexisting with the VCS of the present microservice-architecture. This way, redundancy of code in a team is kept very low, by simultaneously having a managable amount of repositories accross a division of a company. With this approach complexity can not be completely eleminated, but at least mitigated to a certain degree. Moreover having different repositories for different teams in an organistaion will simplify the use of many accounts for developent and testing, as discussed in a later section.

Looking at tools supporting development, literature is currently facing a lack of tooling, in terms of the variety and maturity provided [Yus+19], [Lei+19]. With a share of around 80% [Lei+19], of practitioners using the Serverless [11] framework, this tool is the most frequently used tool in the market. Besides this framework, Terraform [12] and CloudFormation [13] are mentioned as well in some papers, but theit appearence is far less compared to the previous. Especially in conjunction with the ci cd pipeline, the use of tooling can play an important role. By using one of the above mentioned tools, the deploament is configuered withing those tools, providing an abstraction layer between the present architecture and the providers plattform. Therby thees tools enabels hybrid architectures accross multiple cloud probiders. When striving to gain independence and reduce the effect of vendor lock in [see *benefits and drawbacks*], these frameworks are supporting that process. At this conjunction additional operational should not remain unmentioned. Even though Serverless, Terraform and CloudFormation provide the opportunity to register many accounts form all kinds of servless providers, inculding AWS, Azure, IBM and more, they require in initial load in configuration. ANotehr issue whcih need to be addressed, is that new features offered by a platform wont be adjusted to third parties, which engenders latency time until the feautre mihgt be provided by the third party.

3.5 Abwägung Open-Source und Cloud-Vendor

Wie bereits im vorherigen Abschnitt beschrieben, OpenFaaS

Die Abwägung zwischen der Implementation von Function as a Service in der unternehmenseigenen Cloud bzw. Servern oder dem Outsourcen an einen proprietären Cloud-Vendor, sollte wohl überlegt sein, da hiervon in der Folge eine Vielzahl an Möglichkeiten und Restriktionen abhängt.

Entscheidet man sich für ersteres, also dem Aufbau einer privaten FaaS-Plattform so stehen hierfür, mit IBM Apache OpenWhisk, Fission, OpenFaaS oder Kubless, nahezu identisch viele Frameworks zur Verfügung wie bei den öffentlichen Vendoren. Schaubild CNCF Serverless Landscape ggf. einfügen. Des Weiteren bietet dieser Weg, abgesehen von der Vielfalt an Fra-

[11] https://serverless.com/
[12] https://www.terraform.io/
[13] https://aws.amazon.com/cloudformation/

meworks mit wiederum unterschiedlichen Eigenschaften, die Möglichkeit die Größe der Funktionen oder die maximal zulässige Laufzeit eines Containers individuell anzupassen. Dies würde einerseits eine granularer Verrechnung der in Anspruch genommenen Ressourcen der einzelnen Bereiche oder Teams zulassen, zugleich aber auch ein Operations-Team verlangen, welches sich in das jeweilige Framework einarbeitet, die Plattform zu dessen Betrieb aufsetzt und die spätere Wartung dieser übernimmt. [MPDF+18].

Eine Alternative stellen proprietäre Lösungen von öffentlichen Cloud-Vendoren, wie beispielsweise Amazon Web Services (AWS) Lambda, Microsofts Azure Functions, IBM Cloud Functions oder Google Cloud Functions, bei welchen sich von Unternehmensseite aus niemand um die Wartung der Infrastruktur, die Skalierung der Services, die Behandlung von Fehlermeldungen o.ä. kümmern muss. Diese Aufgaben werden in der Folge von dem Plattformbetreiber übernommen. Natürlich sind diese Dinge in erster Linie Aufgabe des Plattformbetreibers, jedoch haben sie unmittelbare Auswirkungen auf das Unternehmen, sollten Probleme beim Skalieren von Funktionen oder dem Monitoring auftreten. Sollte dies der Fall sein, so sind die Entwickler von den bereitgestellten Debugging Möglichkeiten und Monitoring-Lösungen der Plattform abhängig um Probleme schnellstmöglich zu beheben, sollte dies die Plattform nicht tun. Daneben ist ein weiterer Punkt des Vendor Lock-Ins die von Anbieter zu Anbieter variierende Infrastruktur, welche ein einfaches Shiften von Funktionen erheblich erschwert. Zudem sind die Benutzung der Funktionen meist automatisch mit der Inanspruchnahme weiterer Services der Plattform, wie dem Message Queuing oder der Datenspeicherung, gekoppelt.

Dies hat sowohl Vor- als auch Nachteile. Ist man bei der Einrichtung der privaten FaaS-Cloud auf die unternehmensinternen Ressourcen beschränkt, so bieten die Cloud-Anbieter, neben den Restriktionen des Vendor Lock-Ins, in den meisten Fällen ein großes Ökosystem an weiteren Services, welche sich problemlos an die Funktionen anbinden lassen. Im Folgenden wird daher, auch wenn sich diese Arbeit hauptsächlich auf FaaS beschränkt, das Ökosystem der einzelnen Cloud-Anbieter kurz betrachtet, um einen besseren Überblick über deren zusätzliche Leistungen zu erhalten und eine fundierte Entscheidung treffen zu können.

3.5.1 Vendoren-Analyse

Ein weiterer Punkt bei der Migration von Teilen einer besehenden Architektur, in diesem Fall einer Microservice Architektur, ist die von dem Cloud-Vendor zur Verfügung gestellte Möglichkeit der Orchestrierung der Funktionen. Diese entscheidet darüber wie performant die Funktionen parallel ausgeführt werden können und wie groß der Runtime-Overhead der jeweiligen Plattformen dabei ist. Zudem spielt der Preis der hierfür anfällt eine nicht unerhebliche Rolle, da die sequentielle als auch die parallele Ausführung von Funktionen bei mittleren und großen Softwareprojekten häufig auftreten.

Wie bereits in *Potentiale und Herausforderungen* unter *Statelessness* erwähnt spielt bei der Kopplung von Funktionen die Weitergabe des Anwendungs- bzw. Funktions-*States* und dessen Geschwindigkeit eine wichtige Rolle. Nur durch das richtige Zusammenspiel vieler Funktionen ist es möglich große Anwendungen zu bauen und komplexe Abläufe umzusetzen. Bei der Betrachtung der unterschiedlichen Hosting-Lösungen soll der Fokus auf die beiden am häufigsten genutzen proprietären FaaS-Lösungen [Lei+19], AWS Lambda mit Amazon Step Functions und Microsoft Azure Functions mit Azure Durable Functions, gelegt werden. Die Seite der Open-Source Lösungen wird von IBM OpenWhisk, mit IBM Composer als Orchestrierungs-Lösung, vertreten.

[Lóp+18] hat diese drei Frameworks genuaer untersucht und bei der sequentiell und parallele Planung sowie der Weitergabe des *States* der drei Orchestrierungs-Tool erhebliche Unterschiede feststellen können. Vorab soll an dieser Stelle vermerkt werden, dass nur „warme“ Instanzen für jegliche nachfolgende Tests genutzt wurden, um Ungenauigkeiten, durch variierende *Cold-Start* Zeiten [siehe [Man+18] und [JC18]], vorzubeugen.

In Bezug auf die Ausführung von aufeinander folgenden Funktionen, sequentielle Verarbeitung, erwiesen sich IBM's Composer und ASF als deutlich schneller im Vergleich zu ADF. So Betrug der Overhead, also die Zeit welche nicht zur Ausführung der Funtkion verwendet wurde, bei 40 hintereinander geschalteten Funktionen 1,1s für IBM Composer und 1,2s für AWS Step Functions. Azure Durable Functions brauchten hingegen für die selben 40 Funktionen ganze 8s. Bei weiteren Durchführungen mit [5, 10, 20, 40, 80] stellte sich jedoch heraus, dass IBM Composer die Orchestrierung von Funktionen nur bis zu einer Anzahl von 50 Stück unterstützt. Alles was darüber hinausgeht, müsste durch Orchestrierungs-Tool von Drittanbietern

übernommen werden [Lóp+18]. ADF und ASF sind wiederum in der Lage *Workflows* festzulegen, welche über Tage und Monate lauffähig sind.

Die Evaluierung des Overheads bei parallel geschalteten Funtkionen wurde für ASF und ADF durchgeführt. Dabei wurde wieder mit 5 Funktionen begonnen und sich, wie oben beschrieben, bis auf 80 hochgearbeitet. Die Ergebnisse waren eindeutig. Bei einer Anzahl von 80 Funktionen hatten Azure Durable Functions mit einem durchschnittlichen Overhead von 32.1s fast das doppelte Volumen von AWS Step Functions mit einem durchschnittlichen Overhead von 18.3s. Die Ergbenisse legten zudem nahe, dass ASF zuverlässiger bei der Vorhersage des zu erwatenden Overheads ist als ADF. Microsofts Overhead stieg nicht immer gleichbleibend exponentiell an wie der von Amazon, was eine Prognose über das Verhalten erschwert.

Bei der Evaluierung für die Eignung von parallel geschalteten Funktionen viel IBMs Composer direkt zu Beginn aus dem Testportfolio heraus, da parallele Ausführungen nicht unterstützt wurden [stand 2018 [Lóp+18]]. Mittlerweile [stand 2020] wird von Seiten IBMs die parallele Ausführung von Funktionen seitens Composer zwar unterstützt und gesagt, dass seitens Composer dies nicht auf eine bestimmte Anzahl von Funktionen beschränkt ist [14]. Allerdings wird explizit erwähnt, dass eine Limitierung gleichzeitig ausführbarer Funktionen Seitens OpenWhisk besteht, welche bei Überschreitung zu Fehlern führen kann: „[...] many concurrent invocations may hit OpenWhisk limits leading to failures: failure to execute a branch of a parallel composition or failure to complete the parallel composition [...]“ [15]. Die derzeitige Limitierung gleichzeitiger Funktionen in OpenWhisk liegt bei 100 pro *Namespace* [16].

Zuletzt wurden die drei Orchestrierungs-Lösungen bezüglich der Weitergabe des *Application-States* untersucht. Aufgrund der Begrenzung von ASF auf 32KB wurde bei den beiden anderen Lösungen die selbe Größe gewählt. Dieses mal wurden nur 5 sequentiell ablaufende Funktionen getestet. Die Grenze bei IBM Cloud Functions lag 2018 offiziell bei 1MB, liegt mittlerweile aber bei 5MB [17]. ADF hingegen ermöglich die Weitergabe von bis zu 60KB. Es zeigte sich, dass IBM Composer und AWS Step Functions bei der Ausführung ohne *Payload*, jeweils einen Overhead von 175.7ms und 168.0ms

[14] https://github.com/apache/openwhisk-composer
[15] https://github.com/apache/openwhisk-composer
[16] https://github.com/apache/openwhisk/blob/master/docs/
[17] https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-limits

hatten. Mit Payload betrug der Overhead in ms für Composer 298.4 und Step Functions 287.0, was eine Zunahem von 70% darstellt [siehe Abbildung 6]. Azure Durable Functions stieß bei diesem Test deutlich hervor. Mit einem Overhead von 766.2ms ohne Paylod und 859.5ms mit Payload ist der grundlegende Overhead zwar deutlich höher als bei den beiden vorherigen, steigt unter Last aber nur um 12% an [Lóp+18].

Abbildung 6: Overhead bei 5 sequentiellen Funktionen mit einem Payload von 32KB, nach [Lóp+18]

Platform	**Overhead (ms)**		**Increase (%)**
	Without payload	**With payload**	
IBM Composer	175.7	298.4	70%
AWS Step Functions	168.0	287.0	71%
Azure DF	766.2	859.5	12%

Anhand der durch Lopez et al. gezeigten Ergebnisse lässt sich festhalten, dass AWS mit Step Funtion das ausgereifteste Orchestrierungs-Tool zur Verfügung stellt. Sowohl bei der sequentiellen als auch bei der parallelen Ausführung beitet AWS Lösungen für langlebige und kurzlebige Funktionskopplungen. Zudem ermöglicht die Limitierung des *States* auf 32KB eine klare auskunft über die entstehenden Kosten zu geben. Hat man vor FaaS lediglich für leichtgewichtige Aufgaben zu nutzen, so spielt IBM Composer bei der Kopplung von bis zu 50 Funktionen seine Stärken aus und ist dabei geringfügig schneller als AWS. Für die Festlegung längerer Abläufe, welche auf das Laufen über Tage bis hin zu Monaten ausgelegt sind, ist Composer nicht geeignet [Lóp+18]. Steht der Austausch des *Application-States* zwischen Funktionen im Vordergrund und können höhere Latenzzeiten in Kauf genommen werden, bietet Azure mit einer Kapazität von 60KB eine Alternative zu AWS und IBM. Auch Abläufe die länger Zeit beanspruchen können mit ADF konfiguriert werden. Das Konzept von *async/await* bei sequentiellen und *fan-out/fan-in* [18] bei parallelen Abläufen bietet zusätzlich eine etwas leichtere Umsetztung als AWS und IBM [Lóp+18]. Kein garantie,

[18] https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-cloud-backup

dass der folgende Funktionsaufruf auf die selbe bereits laufende Instanz einer Funktion trifft, welche dann wiederum auch zugriff auf dem im Memory gespeicherten State hat. Es ist daher notwendig, dass der Application-State extern gespeichert wird. Auf diesen sog. "*Shared memory*"müssen in der Folge alle Funktionen drauf zugreifen, wenn Applikations-Daten benötigt werden.

3.5.2 Tools zur Entwicklungsunterstützung

Terraform

IBM Composer, Amazon Step Functions, Azure Durable Functions Serverless Framework

Chalice SAM Serverless Framework

Results

3.5.3 Cross-Functions in Multi-Cloud Lösung

CloudFormation, Terraform or Serverless Framework tool [IS18]

3.6 Auswirkungen auf die Testumgebungen

Zum Testen einer serverlosen Applikation stehen mehrere Möglichkeiten zur Verfügung, welche ihre Vor- und Nachteile haben. Unterschieden wird in lokales Unit-Testen, Canary Release Testen ing sowie A/B Testen und Integrationstests. War OpenWhisk zu Beginne einer der ersten Anbieter, welcher lokale Unit-Tests unterstützte, so zogen AWS mit SAM [19] und Azure mit seinen sog. Function Core Tools [20] nach. In diesen Unit-Tests liegt jedoch bereits eine Einschränkung, die bei der Migration bedacht werden muss. Gibt es bei Anwendungen die nicht über einen *Serverless Cloud-Provider* laufen oft die Möglichkeit Teile der Anwendung durch lokale Kopien von Datenbanken oder Message-Queues, welche denjenigen in der Produktion sehr ähnlich sind, bi dem Testen zu integrieren, so ist dies bei serverlosen Funktionen schwerer [RC17]. Dadurch, dass die Funktionen in den meisten Fällen mit anderen Funktionen sowie Datenbanken und ggf. noch einigen weiteren Services des Anbieters interagieren müssen, ist es nicht möglich dies lokal zu simulieren, zumal in der Plattform gesetzte Konfigurationen lokal nicht umgesetzt werden können. Kennzahlen wie die Ausführungszeit von Funktionen, das Laden von Abhängigkeiten (Libraries) und Verzögerungen durch

[19] $https://aws.amazon.com/de/serverless/sam/?nc1 = h_ls$

[20] https://docs.microsoft.com/en-us/azure/azure-functions/functions-develop-local

Cold-Starts können nicht akkurat wiedergegeben werden [Rac+19]. Mit der Abgabe der Hoheit über die Infrasturktur kann hinzukommend auch nicht mehr der Server bestimmt werden, auf dem die Funktion letzten Endes gestartet wirde, was im zweifel, bei älterer Hardware, die Ausführungszeit der Funktion erhöhen kann. Es ist daher unumgänglich den Service auch auf der Plattform selber zu testen, um zuverlässige Daten über Perfomanz und Kompatibilität zu erhalten.

Eine Möglichkeit dafür bietet das sog. Canary-Testen, bei welchem es nicht erforderlich ist die komplette Servicelandschaft auf einen DEV-Account zu spiegeln. Stattdessen wird eine neue Version einer bereits existierenden Funktion in Produktion geladen und nur ein bestimmter Teil der Nutzer bzw. Tester darauf umgeleitet. In der Praxis werden *Canary-Testing* sowie *A/B-Testing* aber nicht sehr häufig durchgeführt [Lei+19], da dies negative Auswirkungen auf die Performance der in Produktion laufenden Anwendung haben kann. Vor allem Lasttests können dafür sorgen, dass es zu spürbaren Latenzen bis hin zu Ausfällen bei den Nutzern kommt.

Es bietet sich daher an, den Aufwand einer Spiegelung zu betreiben, da so die in Produktion laufenden Funktionen nicht von Tests beeinflusst werden. Viele Anwender haben dies bereits umgesetzt und es scheint sich als „Best Practice“ plattformunabhängig etableiert zu haben [Lei+19]. An sich ist dies jedoch nicht verwunderlich, da FaaS bzw. Serverless bei diesem Vorgehen seine Stärken ausspielen kann. So ist es preislich unabhängig, ob das Service-Ökosystem inkl. Funktionen gespiegelt wird oder die Test neben den in Produktion genutzten Funktionen durchgeführt werden.

wird daher in der Praxis häufig die Funktionene in Produktion zu Testen bzw. einen gespiegelten Testaccount der eigentlichen, in Produktion laufenden, serverlosen Anwendungslandschaft anzulegen.

So muss sich meistens auf lokale Unit test beschränkt werden. Um Komponenten bzw. Funktionen einem Integrationstest zu unterziehen, muss sich genau über die Möglichkeiten bei dem jeweiligen Cloud provider informiert werden, da Preise und infrasturkurelle Gegbenheiten variieren. AWS bietet über

SAM Für traffic shifting

Dies ist teilweise Vendorenspezifisch und bedarf der auseinandersetzung

3.7 Konsequenzen für das Monitoring

4 Anwendung des Leitfadens

4.1 Auswertung der Ergebnisse

4.1.1 Beurteilung der Kollaborationsauswirkungen

4.1.2 Beurteilung der Stabilität

4.1.3 Beurteilung der Skalierbarkeit

4.1.4 Beurteilung des Monitorings

4.1.5 Beurteilung der Testmöglichkeiten

4.2 Korrektur und Anpassungen

5 Abschließende Betrachtung

5.1 Absehbare Entwicklungen

[AAS19] Serverless immer mehr bekanntheit und mehr Papers etc. ... Später market für funktionen die optimiert sind etc. [SKM20]
Real time communication tool [SKM20]
Real time tracking gps [SKM20] da beide nicht auf dem Application state benötigen/ beruhen [SKM20]
[Hel+18]

5.2 Zusammenfassung

5.3 Weiterführende Forschung

Serverless Computing: A Survey of Opportunities, Challenges and Applications store for functions [SBW19]

Literaturverzeichnis

[Wri+98] Andrew Wright u. a. "Compiling Java to a typed lambda-calculus: A preliminary report". In: *International Workshop on Types in Compilation*. Springer. 1998, S. 9–27.

[FK03] Ian Foster und Carl Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.

[Fox+09] Armando Fox u. a. "Above the clouds: A berkeley view of cloud computing". In: *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS* 28.13 (2009), S. 2009.

[Liu+11] Fang Liu u. a. "NIST cloud computing reference architecture". In: *NIST special publication* 500.2011 (2011), S. 292.

[MG+11] Peter Mell, Tim Grance u. a. "The NIST definition of cloud computing". In: (2011).

[Reg11] Antonio Regalado. "Who coined 'cloud computing'". In: *Technology Review* 31 (2011).

[Bal+17] Ioana Baldini u. a. "Serverless computing: Current trends and open problems". In: *Research Advances in Cloud Computing*. Springer, 2017, S. 1–20.

[Eiv17] Adam Eivy. "Be wary of the economics ofSServerless"Cloud Computing". In: *IEEE Cloud Computing* 4.2 (2017), S. 6–12.

[Fox+17] Geoffrey C Fox u. a. "Status of serverless computing and function-as-a-service (faas) in industry and research". In: *arXiv preprint arXiv:1708.08028* (2017).

[MB17] Garrett McGrath und Paul R Brenner. "Serverless computing: Design, implementation, and performance". In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE. 2017, S. 405–410.

[PS17] Hussachai Puripunpinyo und MH Samadzadeh. "Effect of optimizing Java deployment artifacts on AWS Lambda". In: *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE. 2017, S. 438–443.

[RC17] Michael Roberts und John Chapin. *What Is Serverless?* O'Reilly Media, Incorporated, 2017.

[SD17] Josef Spillner und Serhii Dorodko. “Java code analysis and transformation into AWS lambda functions”. In: *arXiv preprint arXiv:1702.05510* (2017).

[BRH18] Daniel Bardsley, Larry Ryan und John Howard. “Serverless Performance and Optimization Strategies”. In: *2018 IEEE International Conference on Smart Cloud (SmartCloud)*. IEEE. 2018, S. 19–26.

[Ben+18] Alexander Benlian u. a. “The transformative value of cloud computing: a decoupling, platformization, and recombination theoretical framework”. In: *Journal of management information systems* 35.3 (2018), S. 719–739.

[Fow18] Martin Fowler. “Serverless Architectures”. In: *martinfowler.com* (2018). https://www.martinfowler.com/articles/serverless.html.

[Hel+18] Joseph M Hellerstein u. a. “Serverless computing: One step forward, two steps back”. In: *arXiv preprint arXiv:1812.03651* (2018).

[IMS18] Vatche Ishakian, Vinod Muthusamy und Aleksander Slominski. “Serving deep learning models in a serverless platform”. In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2018, S. 257–262.

[IS18] Vitalii Ivanov und Kari Smolander. “Implementation of a DevOps pipeline for serverless applications”. In: *International Conference on Product-Focused Software Process Improvement*. Springer. 2018, S. 48–64.

[JC18] David Jackson und Gary Clynch. “An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions”. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE. 2018, S. 154–160.

[KS18] Kyriakos Kritikos und Paweł Skrzypek. “A review of serverless frameworks”. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE. 2018, S. 161–168.

[Lóp+18] Pedro García López u. a. “Comparison of faas orchestration systems”. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE. 2018, S. 148–153.

[Man+18] Johannes Manner u. a. “Cold start influencing factors in function as a service”. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE. 2018, S. 181–188.

[MPDF+18] Sunil Kumar Mohanty, Gopika Premsankar, Mario Di Francesco u. a. “An Evaluation of Open Source Serverless Computing Frameworks.” In: *CloudCom*. 2018, S. 115–120.

[SS18] Mohit Sewak und Sachchidanand Singh. “Winning in the era of serverless computing and function as a service”. In: *2018 3rd International Conference for Convergence in Technology (I2CT)*. IEEE. 2018, S. 1–5.

[Adi+19] Paarijaat Aditya u. a. “Will Serverless Computing Revolutionize NFV?” In: *Proceedings of the IEEE* 107.4 (2019), S. 667–678.

[AAS19] Mohammed Al-Ameen und Josef Spillner. “A systematic and open exploration of FaaS research”. In: *ESSCA, Zurich, Switzerland, December 21, 2018*. CEUR-WS. 2019, S. 30–35.

[Bro19] Nicolas Brousse. “The issue of monorepo and polyrepo in large enterprises”. In: *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*. 2019, S. 1–4.

[Cas+19] Paul Castro u. a. “The server is dead, long live the server: Rise of Serverless Computing, Overview of Current State and Future Trends in Research and Industry”. In: *arXiv preprint arXiv:1906.02888* (2019).

[Jon+19] Eric Jonas u. a. “Cloud programming simplified: A berkeley view on serverless computing”. In: *arXiv preprint arXiv:1902.03383* (2019).

[Kap19] Ayhan Kaplan. “Framework for migrating deployed serverless applications”. Magisterarb. 2019.

[Lei+19] Philipp Leitner u. a. “A mixed-method empirical study of Function-as-a-Service software development in industrial practice”. In: *Journal of Systems and Software* 149 (2019), S. 340–359.

[Rac+19] Louis Racicot u. a. “Quality Aspects of Serverless Architecture: An Exploratory Study on Maintainability”. In: *Proceedings of the 14th International Conference on Software Technologies, ICSOFT*. 2019, S. 26–28.

[SBW19] Mohammad Shahrad, Jonathan Balkind und David Wentzlaff. “Architectural implications of function-as-a-service computing”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, S. 1063–1075.

[VE+19] Erwin Van Eyk u. a. “The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms”. In: *IEEE Internet Computing* (2019).

[Yus+19] Vladimir Yussupov u. a. “A Systematic Mapping Study on Engineering Function-as-a-Service Platforms and Tools”. In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2019)*. ACM, Dez. 2019, S. 229–240. DOI: 10.1145/3344341.3368803.

[SKM20] Hossein Shafiei, Ahmad Khonsari und Payam Mousavi. “Serverless Computing: A Survey of Opportunities, Challenges and Applications”. In: (2020).