

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 erzielen. 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 Unternehmenn, 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.

Index of abbreviations

| | |
|--------|--|
| ADF | Azure Durable Functions |
| API | Application Programming Interface |
| 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 |
| CLI | Command Line Interface |
| CNCF | Cloud Native Computing Foundatino |
| CPU | Central Processing Unit |
| CRD | Custom Resource Definition |
| DevOps | Development and Operations |
| DEV | Development |
| DSL | Domain Specific Language |
| FaaS | Function as a Service |
| HPA | Horizontal Pod Autoscaler |
| IAM | Identity Access Management |
| IaaS | Infrastructure as a Service |
| IaC | Infrastructure as Code |
| INT | Integration (Testing) |
| IT | Information Technology |
| JVM | Java Virtual Machine |
| NIST | National Institute of Standards and Technology |
| OOP | Object Oriented Programming |
| OS | Operating System |
| PaaS | Platform as a Service |
| PR | Pull Request |
| PRD | Production (Environment) |
| 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 Einordnung dieser beiden Konzepte in die Infrastruktur des Cloud Computings⁸ [mell2011nist], ist dieser Prozess noch im Gange, wobei mittlerweile die unterschiedlichen Definitionen der öffentlichen 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 [regalado2011coined].

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 [fox2009above], so erlangte es erst ab 2006 einen immer größer 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 [fox2009above]. Des Weiteren wurde 2009 in einer Berkeley-Studie die sechs größten Potentiale des Cloud-Computings herausgearbeitet welche bis heute in den unterschiedlichen XaaS-Konzepten umgesetzt wurden. Im folgenden sind diese kurz aufgelistet.

1. Unbegrenzte Ressourcen wann immer sie benötigt 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 [mell2011nist]. 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.

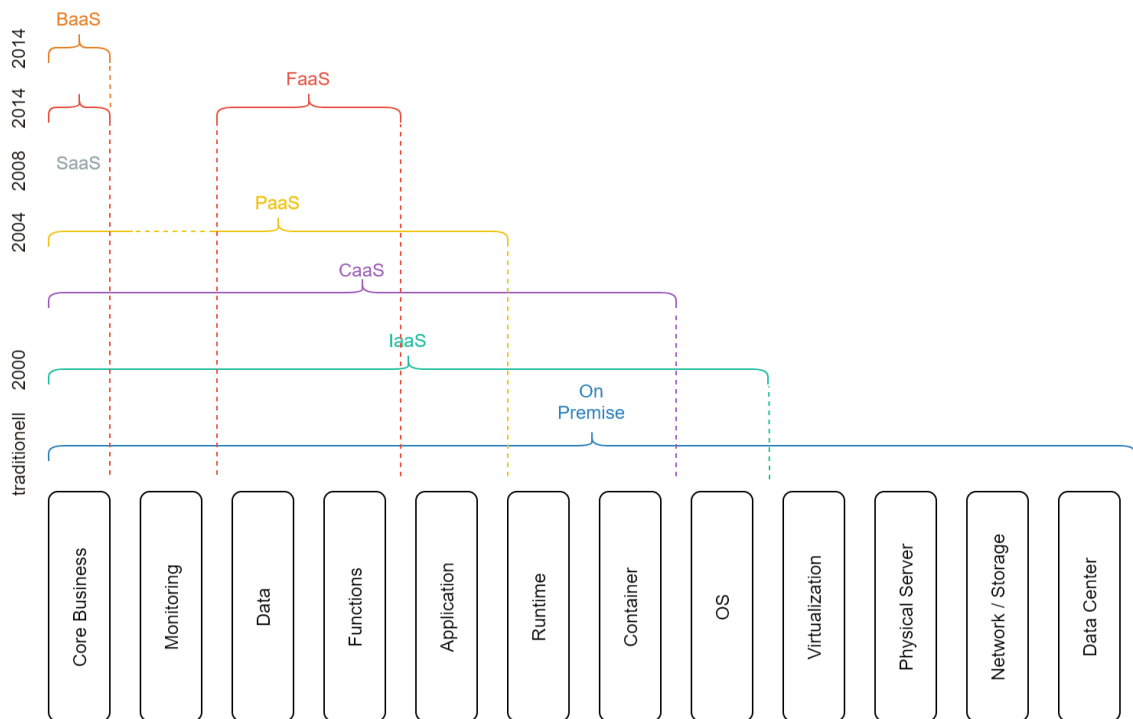
“Cloud Computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.” [mell2011nist].

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 Charakteristiken, 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 Cloud 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 einer dritten Partei gehostet werden. Mit dem Deployment auf eine *Community cloud* teilen sich verschiedene Unternehmen eine Cloud, welche entweder von ihnen gemeinsam, 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 Unternehmen die Räumlichkeiten bzw. Ressourcen von einem Drittanbieter nutzt. Das letzte Deployment-Model stellt die sog. *Hybrid cloud* dar, welche eine Kombination aus den vorherigen darstellt [mell2011nist].

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 ?? 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 wurde¹. 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 Anbieter findet sich online von der CNCF²

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 [mell2011nist] der Fall ist. Die Anbieter sind sich jedoch über die Funktionalität von FaaS größtenteils einig, was das Spektrum an Funktionalitäten angeht. So definiert Microsoft Azure Functions als: „[...] event-driven serverless

¹<https://docs.aws.amazon.com/lambda/latest/dg/lambda-releases.html>

²<https://github.com/cncf/wg-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 hinter 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 Anbieters 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 Infrastruktur des Anbieters angepasste, Trigger zur Verfügung. Diese ergeben sich meist 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 SNS, Google Pub/Sub etc.] auftreten können.

Abbildung 2: ⁴

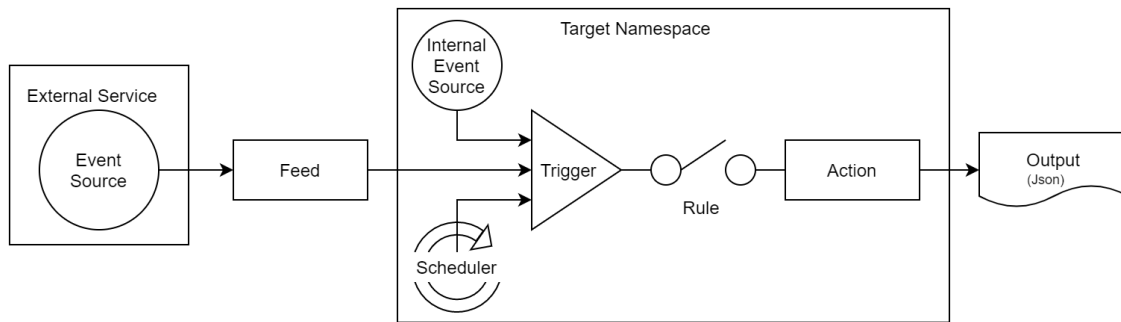


Abbildung ?? gibt hierbei eine Übersicht über die verschiedenen Event-Trigger, welche derzeit bei den großen, proprietären Anbietern (AWS, Google, Microsoft) existieren. Da hinter OpenWhisk mit IBM ein großes Unternehmen steht, wird dieses Modell in der Literatur häufig als Referenz in Bezug auf die anderen Anbieter verwendet. Es kann angenommen werden, dass andere Anbieter ein ähnliches Konzept bei ihrer FaaS Architektur verfolgen, da Apache OpenWhisk in einer großen Cloud (IBM) bereitgestellt wird [van2019spec]. 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 benötigten Ressourcen, wie er es bei PaaS (wobei es hier variierende und bereits granularere Modelle gibt) oder IaaS der Fall ist, sondern nur für die tatsächlich genutzten Ressourcen.

Grundsätzlich ist zwar der Preis der für das Laufen einer einminütigen Funktionsausführung verglichen mit dem Laufen des selben Codes auf einem von den Ressourcen her ähnlich bestückten PaaS-Server günstiger [jonas2019cloud], 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 gelöst werden, bei denen der Nutzer nur für die Ressourcen bezahlen muss, die die Funktion bei dem Aufruf nutzt und nicht für jene, die ohnehin einkommende Aufrufe provisioniert wurden.

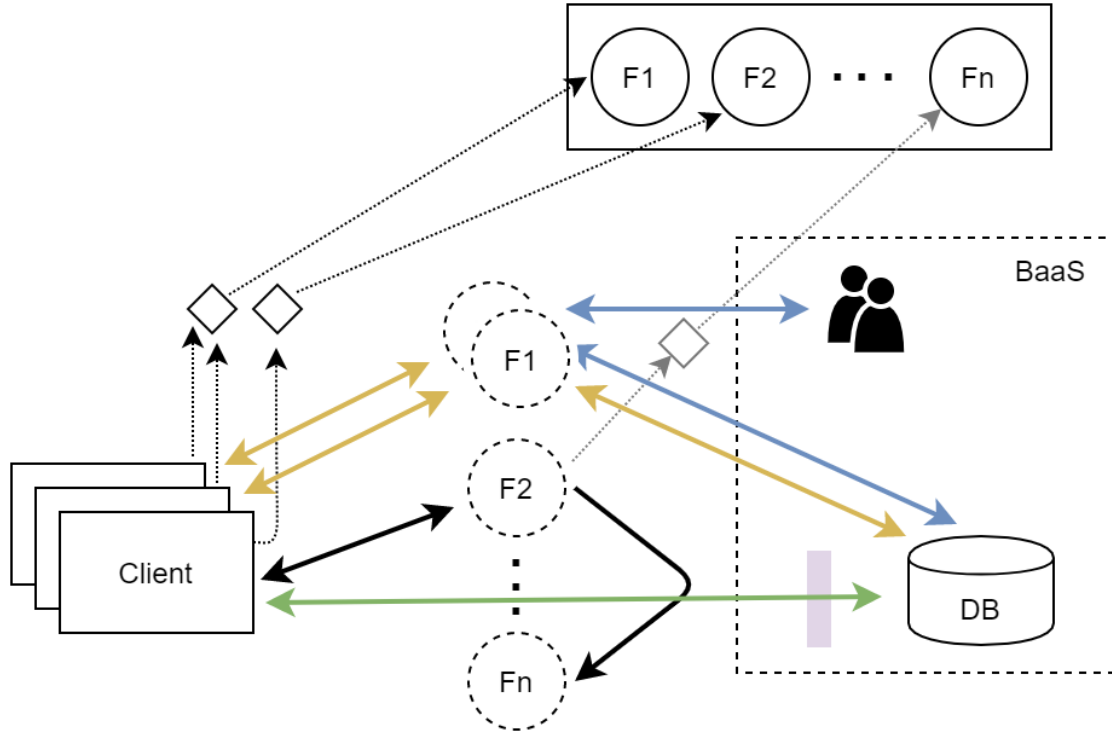
³<https://azure.microsoft.com/en-us/services/functions/>

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 Anbieters, und ist dem Nutzer nicht zugänglich, zumindest bei den proprietären Anbietern. Letzteres ist zwar keine einzigartige Eigenschaft von FaaS, da dieses Konzept bereits lange bekannt ist und auch bei der Microservice-Entwicklung 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 an Flexibilität bei der Entwicklung der Anwendung einräumt. Eine weitere häufig in der Literatur referenzierte Definition ist [fowler2018serverless], 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 Vendor Lock-In dieser jedoch weder als schlecht noch gut zu bewerten ist und vielmehr nüchtern auf seine Vor- und Nachteile analysiert werden sollte. verbunden, da dieser Entscheidungen über die Infrastruktur trifft, mehr dazu in *Potentiale und Herausforderungen*.

Abbildung ?? 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 Funktionen lädt. Wird eine Funktion von einem Client aufgerufen, so wird bei Aktivierung eines Triggers eine Kopie der jeweiligen Funktion instantiiert und alle mit dieser Funktion verbundenen Abhängigkeiten geladen (bspw. benötigte Module). Je nachdem wie frequentiert die Funktionen von den Clients angefordert werden, werden die vorhandenen Instanzen automatisch von dem Provider

⁵<http://openwhisk.apache.org/>

Abbildung 3: Anwendung mit FaaS und (P)BaaS, angelehnt an [shafiei2020serverless]



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 Infrastruktur 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 [liu2011nist], [foster2003grid] 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 entwickle [leitner2019mixed]. Erste Vorschläge wie die Referenzarchitektur von FaaS aussehen könnte, haben sich mit [van2019spec] 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 adressiert worden, so stieg die Präsenz von hier an kontinuierlich bis 2019, in Journals, Konferenzen und Workshops an [Yussupov2019'SystematicMappingStudyFaaS].

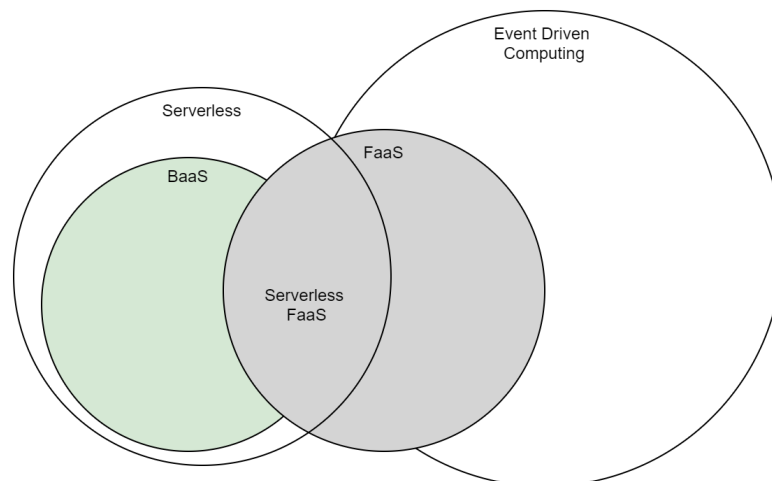
2.3 Abgrenzung von FaaS zu Serverless

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 [jackson2018investigation]. Wie so häufig ist die Abgrenzung bzw. die Unterscheidung bei sich neu entwickelten 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. ??], um serverseitige Aufgaben wie Datenbankabfragen, Nutzerverwaltung o.ä. zu regeln [fowler2018serverless].

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 [baldini2017serverless]. Jedoch geht mit der Abgabe an Zuständigkeiten auch ein gewisser Vendor Lock-In einher, wobei die Anbieter darauf achten, das mit FaaS eine möglichst große Zahl der in ihrem Service-Ökosystem vorhandene Services genutzt werden [kritikos2018review]. 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 [leitner2019mixed] in 58% der Fälle, mit FaaS gleichgesetzt, was eine Abgrenzung erschwert. Abbildung ?? verdeutlicht in welcher Beziehung die unterschiedlichen Konzepte stehen.

Abbildung 4: Serverless Concept, including FaaS and BaaS



Besteht ein 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 [leitner2019mixed]. Bei letzterem fungieren die Funktionen 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 argumentieren, dass ein Großteil der infrastrukturellen 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 Plattform bereitgestellt, womit sich die Entwickler immer noch um die Anwendungsstruktur kümmern müssen [kaplan2019framework]. In FaaS hingegen wird dieser Teil übernommen und der Entwickler konzentriert 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 gleichzusetzen, wie suggeriert wird [fowler2018serverless]. Es obliegt weiterhin den Entwicklern qualitativ hochwertigen Code zu schreiben und diesen in der Cloud-Umgebung zu testen, um dessen Performanz sicherzustellen. Die Komplexität entfällt daher nicht vollkommen, sondern verlagert sich zu einem gewissen Teil [eivvy2017wary]. Schaut man sich beispielsweise AWS an, so muss eine Vielzahl an Konfigurationen vorgenommen werden. Zwar kann später eine Vorhandenheit auf alle Funktionen angewendet werden, jedoch ist diese in der Folge von essentieller Bedeutung in einer riesigen „Multitenancy“-Umgebung.

2.4 Potentiale und Herausforderungen

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

Potentiale

Kosteneffizienz

Time to Market / Lead Time Development

DevOps

Testing

Optimale Auslastung von Rechenzentren Greener Computing [shafiei2020serverless]
[fowler2018serverless]

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 Datenabhängigkeit 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 [shafiei2020serverless]. 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 [shafiei2020serverless].

„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“ Zugriffsbeschränkungen umsetzen. Die Services sind von dem jeweiligen Vendor vorgegeben und können bspw. nicht nach dem „Backend For Frontend“-Muster⁶ auf die entsprechenden Client-Typen, wie Tablet, Mobile-Phone oder Desktop angepasst werden. 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 [fowler2018serverless]. Mit FaaS kann dieser Effekt gemindert werden, indem ein leichtgewichtige Logik in form von Funktionen serverseitig umgesetzt wird. Hierbei erfolgt die Interaktion mit den unterschiedlichen 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 dass es zu Inkonsistenzen kommt. Bedingt durch die kurze Laufzeit der Funktionen ist es essentiell, dass auch der Austausch in entsprechender Geschwindigkeit erfolgt. Wie in ei-

⁶<https://samnewman.io/patterns/architectural/bff/>

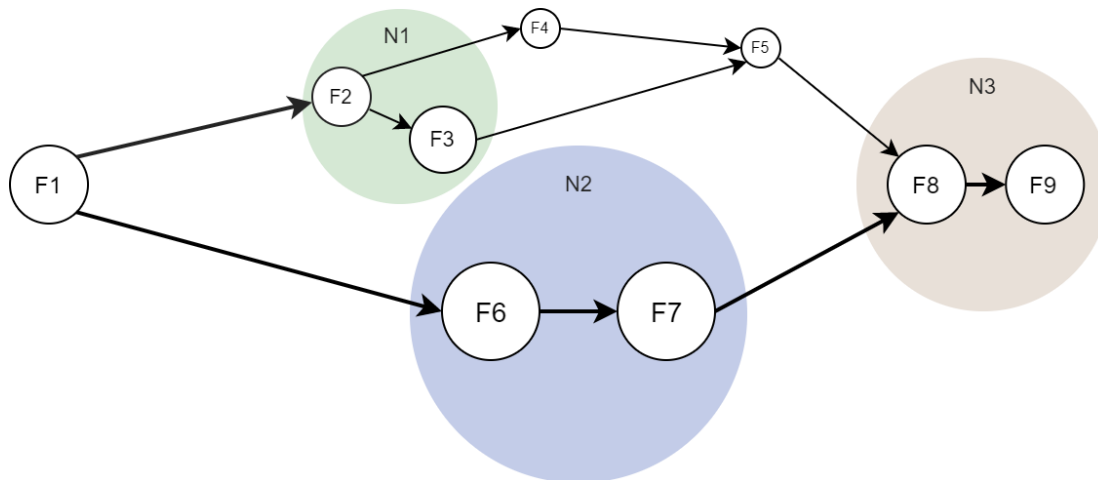
nem Report von Berkeley [jonas2019cloud] herausgearbeitet, erweist sich das schnelle und exakte „State-Sharing“ jedoch immer noch als problematisch dar, betrachtet man die Geschwindigkeit von „Serverless“-Anwendungen im Vergleich mit „Serverful“-Anwendungen.

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 teuer sehr teuer 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 [jonas2019cloud]. 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 Informationsaustausch der Funktionen mit einem externen Speichermedium, kann es bei einer Kopplung von Funktionen aus performancetechnischen Gründen sinnvoller sein, den aktuellen Programmstatus direkt an die nächste 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 Infrastruktur, 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. [shafiei2020serverless] 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 [shafiei2020serverless]



Wenn jeder Nutzer bestimmen könnten auf welcher Node, bspw. Node-X, seine Funktionen laufen sollen, um den Programmstatus 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 Nutzerfunktionen 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 ?? zu sehen würde dieses Problem umgangen werden. Der *Load-Balancer* des Anbieters könnte unbeeinträchtigt 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 beeinflussen kann [manner2018cold], als jene durch die Lokation bedingt [aditya2019will] [jackson2018investigation], 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 höheren 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 Zugriffsbeschränkungen 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 [fowler2018serverless], ist weiterhin Vorsicht geboten. Es nichts an der Tatsache, dass auch sie sich mit Sicherheitsaspekten, der Robustheit und Performance ihrer Infrastruktur ständig weiter auseinander setzen 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 Containersicherheit muss dauerhaft gegeben sein und durch sorgfältiges *Security Managament* sichergestellt werden [mcgrath2017serverless].

Testing

Cold-Starts

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>

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 vorbereitet 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 [shahrad2019architectural]. Ein Container muss auf einer Node gestartet werden und die Funktion, sowie deren Abhängigkeiten geladen werden. Die Zeit die es dauert, bis die Instanz bereit ist hängt dabei von verschiedenen Faktoren, wie der Programmiersprache, der benötigten Bibliotheken (*Libraries*), der Größe der Funktion (menge an Code) und der Hardware [shafiei2020serverless] [jonas2019cloud] 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 [roberts2017serverless]), 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 [roberts2017serverless].

Um dem entgegenzuwirken schlägt [castro2019server] 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, eliminiert werden. Lediglich die größe der Funktionen und deren *Dependencies* müssen noch geladen werden. [ishakian2018serving] 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 Unternehmen zwischen den Mehrkosten und dem Performancezuwachs abzuwägen.

3 Guidline on Migration

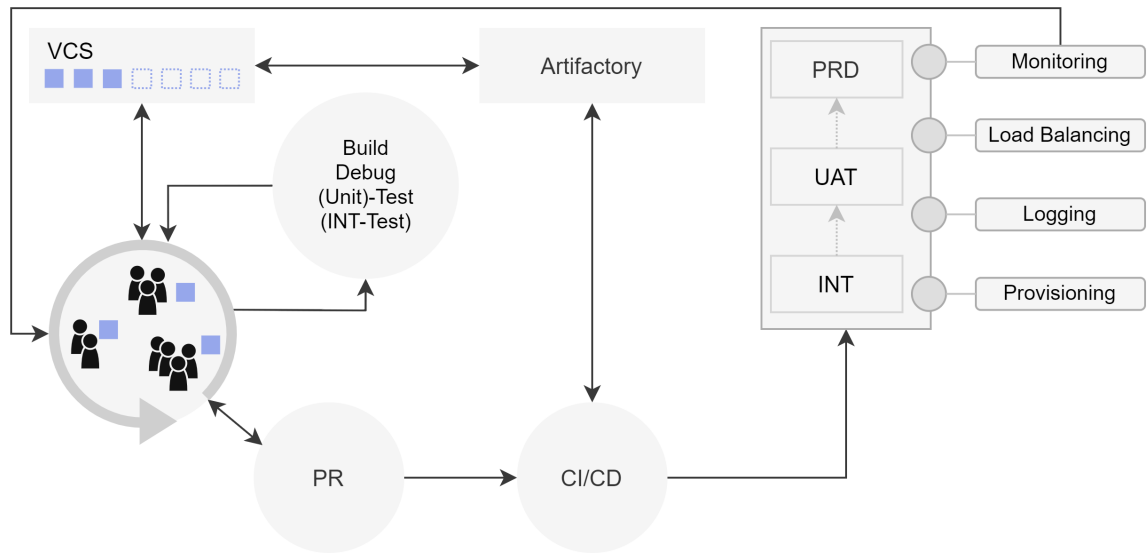
The below-presented guideline for migrating parts of existing microservice architecture to the relatively new cloud-computing concept, Function as a Service, is structured as follows. At first, the decision to migrate is questioned by challenging the motives of adopting this new technology and giving 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 Unternehmen, 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.

Abbildung 6: DevOps Pipeline



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 start-up time of the container the function runs in [manner2018cold]. Especially with Python, Nodejs, and Java [puripunpinyo2017effect], 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 minutes⁹. Later on, when comparing cloud providers with open-source frameworks, there will be guidance on finding 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 [jonas2019cloud]. 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 [jonas2019cloud]. Therefore, services with various workloads, having eventually high peaks 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, deal with concurrency setting¹⁰, 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“ [eivy2017wary], 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 [[sewak2018winning], [leitner2019mixed]]. 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

⁹<https://aws.amazon.com/de/lambda/>

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

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 [**bardsley2018serverless**]. 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. [**wright1998compiling**], should be provided with the method signature. This way, 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 [**spillner2017java**]. Considering these additional steps of migrating a service, written in an object OOP language [**leitner2019mixed**], 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 [**leitner2019mixed**], 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 and tooling

When a microservice architecture is prevalent, FaaS can be adopted with fewer changes than a monolithic architecture [**fox2017status**]. With FaaS, concepts like agile development, continuous delivery and continuous integration, as well as a different mindset amongst the development team, are probably more common. Nevertheless, FaaS goes a step further, then PaaS does, which suggests the assumption that the PaaS-adapted development structure will change. Starting with agile development and „DevOps“, Function as a Service forces companies and teams to adopt an agile mindset and agile techniques [**benlian2018transformative**].

The necessity of iterative cycles is accounted for by the concepts of greater modularity compared to microservices. Persuing procedures from older waterfall driven projects are still prevalent, would not satisfy the core-features of Serverless, respectively, FaaS. With a significant reduction in lead time development, long periods of requirement engineering will hinder the effectiveness of FaaS. Teams that already incorporated agile process frameworks, such as scrum, will have an advantage over those remaining to acquire experience. Serverless development will enhance the importance of continuous delivery and continuous integration of integrating procedures like unit testing and integration testing, to diminish time to market to a minimum. Depending on the size and complexity, which should be kept to a minimum as well, the chances are that FaaS will be considerably faster than the current microservice development.

Furthermore, the size of the teams, to be more accurate, 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 further reduce complexity. Also, reducing complexity and focussing on a specific functionality can help to optimize each function further. By doing so, besides the reduction in startup latency, with the shorter execution time, the platform will charge less [shafei2020serverless].

However, when maintaining a microservice infrastructure, each service and each version is likely to have a separate repository. With Function as a Service, this approach will be challenged regarding the number of functions that make up the application or service. Having a repository for each service, the odds are in favor of experiencing an overhead in version control, maintaining the landscape. Hence teams pursuing this approach are facing duplication in their code base, because the transparency of already existing functions, can not be gathered with little effort [racicot2019quality]. This complexity and duplication seem to be the cause of why many users follow a mono-repository approach over having many small repositories [brousse2019issue].

With that said, FaaS seems to be conflicting with itself to some extent. Although its modularity suits the poly-repository approach quite well, the accompanying complexity, while an application keeps growing, can cause redundancy and inefficiency. Factors that influence the modifiability are the amount of duplicated code, the complexity of the code units and the coupling between the modules [racicot2019quality]. As a suggestion, there should be a mono-repository-approach for each (scrum)-team and a ploy-repo approach for each division. Both would coexist with the VCS of the present microservice-architecture. This way, redundancy of code in a team is kept very low, by simultaneously having a manageable amount of repositories across a division of a company. With this approach, complexity can not be eliminated, but at least mitigated to a certain degree. Moreover, having different repositories for different teams in an organization will simplify the use of many accounts for development and testing, as discussed in a later section.

Literature is currently facing a lack of tooling, in terms of the variety and maturity provided [[Yussupov2019'SystematicMappingStudyFaaS], [leitner2019mixed]]. With a share of around 80% [leitner2019mixed] of practitioners using the Serverless¹¹ framework, this tool is the most frequently used in the market. Besides the Serverless Framework, Terraform¹² and CloudFormation¹³ are mentioned as well, but their appearance is far less to

¹¹<https://serverless.com/>

¹²<https://www.terraform.io/>

¹³<https://aws.amazon.com/cloudformation/>

the former. Especially in conjunction with the CI/CD pipeline, the use of tooling can play an important role. By using one of the tools mentioned above, the deployment is configured withing those tools, providing an abstraction layer between the contemporary architecture and the provider's platform. Thereby these tools enable hybrid architectures across multiple cloud providers. By striving to gain independence and reduce the effect of vendor lock-in [see *benefits and drawbacks*], these frameworks are supporting that process. Especially the Serverless Framework, thanks to its large amount of plugins, provides e.g. an offline simulation of AWS Lambda and the AWS Gateway.

Within this conjunction, additional operational tasks should not remain unmentioned. Even though Serverless, Terraform, and CloudFormation provide the opportunity to register many accounts from all kinds of serverless providers, including AWS, Azure, IBM and more, they face an initial load in configuration purposes. Another issue that needs to be addressed is that new features offered by a platform will not be adjusted to third parties, which engenders latency time until the third party might provide the feature. Thus, these IaC frameworks do not come for free¹⁴. They can be used for free to a certain extent, but when it comes to more advanced tasks, they will charge the user. Finally, they do as well represent some kind of vendor lock-in.

Depending on the purpose of multi-cloud solutions, there might be higher costs than using just one provider. If functions from one provider need to access another's providers function, its functions will be affected by cold starts. Moreover, the physical distance between the two data centers will affect latencies. If the primary purpose of deploying an application to different cloud providers is to have redundant systems in case one fails, the cost will not be affected too much. Still though, additional operational tasks are required to guarantee a stable backup. Testing needs to be done not only on one but on all cloud providers, the application is mirrored to, which will cause additional costs besides the operational overhead of monitoring and testing.

Regarding the deployment pipeline, in practice, a lot of tasks are handled via shell script or any other kind of scripts. Introducing serverless, special tooling for deploying a function to the desired platform is necessary and needs to be included into the present pipeline. If not using one of the IaC frameworks mentioned above, the providers specific CLI has to be integrated, in the case of AWS Lambda CloudFormation. Integration can vary depending on the provider but is inevitable according to practitioners [ivanov2018implementation].

3.5 Balancing between Open-Source and Cloud-Vendor

Choosing an open-source over a cloud vendor, or the other way round, depends on the purpose of utilization. Neither is the so-called vendor lock-in a drawback nor a limitation, compared to an open-source solution; it is solely a tradeoff between the benefits of two different approaches of the same model. Nevertheless, the decision on one of them will affect the possibilities in the process of development and migration. To answer the question on which one to choose, the following things need to be considered.

If the primary purpose is to start rapid development with not having to configure any in-

¹⁴<https://serverless.com/pricing/>

infrastructure, a cloud vendor seems to be a promising solution, especially when a project has no legacy application that needs to be refactored or rewritten. Which of the vendors exactly is to be taken, is dealt with in the following section.

With Amazon Web Services (AWS) Lambda, Microsofts Azure Functions, IBM Cloud Functions or Google Cloud Functions, each of the large serverless providers has a considerable amount of services in its ecosystem. Moreover, the ecosystem is utterly compatible with itself, making the need to find a custom workaround for processes like CI/CD obsolete. AWS and Azure even provide a premade solution for that purpose. E.g. an application requires user authentication, different types of databases, messaging services and hosting, each of the platforms mentioned above can satisfy these needs. Adding to that, they provide an appealing pricing model with free contingents each month, which makes them a feasible candidate for experimenting with different cloud service

Unfortunately, these benefits do not come without restrictions. By ceding the responsibility of provisioning, load-balancing, scaling, error-handling, monitoring and many more duties to the cloud provider, one is dependent on the tooling provided by the platform. Just because the responsibility of maintenance and provisioning is outsourced to the provider, this does not mean that there is no need for an additional monitoring practice, as later stated in *implications on monitoring*. In case something goes wrong on the platform, it is recommended to have, besides the tools offered by the provider, additional information to reproduce the error, find its cause and solve the problem. Another point that has to be considered is the outage of the platform, which is per se not exclusively a cloud provider issue but something that might be dealt with. A possible solution to this is a framework sitting between development and the platform, enabling multi-cloud application that can produce relief.

Depending on how one looks at it, the services can be depicted as an advantage or disadvantage. Even though the different services undertake many tasks, for many tasks, additional services need to be consulted. In the case of AWS, the platform does scales, monitors and sends messages, but the services needed to do so are more than a view.

If the concept of FaaS should be introduced to a company or project along with staying in control of the entire infrastructure, a cloud provider solution is not applicable. In this case, one of the large numbers of open source providers needs to be chosen. If the present application is built upon Kubernetes, for the purpose of virtualization, Kubeless, OpenWhisk and Fission are feasible FaaS options, that can be integrated with little effort [**palade2019evaluation**]. Using an open-source provider, configurations in terms of computing time and resource capacity can be customized, due to being in control of the platform. Introducing FaaS to an upfront ecosystem will increase its performance and efficiency by freeing unused resources. At the same time, and operations teams will be needed that is in charge of implementing a monitoring solution and configuring the properties of the system [**mohanty2018evaluation**].

3.5.1 Vendor analyses

Depending on the decision made in the previous section, there are various frameworks and platforms at choice. Below, three solutions for each approach are presented. Starting with open source frameworks, Fission, Kubeless, OpenFaaS and OpenWhisk will briefly be intro-

duced in the following.

Kubeless is a Kubernetes native serverless framework, being a straight forward extension for a Kubernetes API-Server. Kubeless itself has no API-Server and no data store to store objects since it is reusing everything from Kubernetes. For scaling, Kubeless makes use of the horizontal Pod-Autoscaler, for monitoring Prometheus is used and for deployment via the API, ConfigMaps and Ingress are utilised¹⁵. Thus being built on top of Kubernetes, making use of its CRDs¹⁶ to create functions, users familiar with the Kubernetes-API can process as usual. The language-runtime itself is encapsulated in a container image which will be set in the Kubeless configuration. A list of supported runtime can be found on github¹⁷. To provide a functionality, each function will be deployed in a separate Pod, supervised by Kubernetes HPA. When a function is not used, the HPA will kill the pod to free capacity. Prerequisite is the deployment of the function with a CPU request limit. Depending on the runtime, metric data can be monitored by Prometheus.

Fission is a serverless framework build on top of an underlying Kubernetes infrastructure. Just like Kubeless, Fission does not inherit a resource store and therefore depends on communicating with the Kubernetes API service¹⁸, in order to get access to secrets and „ConfigMaps“. Supporting a large number of programming languages¹⁹, with Fission, CPU and memory resource limits can be configured. Two aspects which make Fission quite unique compared to the other two, is for one thing its ability to define the number of pre-warmed pods and for another thing its richness of possessing a Workflow-System. Whereas the pool of prewarmed pods can be utilized to mitigate the cold start issue, the Workflow-System makes it easier to handle an increase in functions and execute more complex workflows [see **[kritikos2018review]**]. In contrast, Kubeless uses event chaining to create workflows what makes it harder to maintain the workflow graph

OpenFaaS²⁰ provides an so called *faas-provider* interface, letting it integrate with any provider that has implemented the interface. Currently these providers are Kubernetes, Docker Swarm and faas-memory²¹. Each function will be packaged into a docker image and deployed via the OpenFaaS CLI. With its API-Gateway, OpenFaaS can respond to any kind of event and by making use of its auto scaling functionality the min/max amount of replicas can be configured²².

Lastly OpenWhisk is a serverless framework which has initially been developed by IBM and is now backed by Apache. Because OpenWhisk was backed by a large company, its application model is often referenced in literature when the application model of other large cloud providers is indicated. It can be assumed, that other vendors are using a similar concept due to OpenWhisk being provided in a large cloud environment [**van2019spec**]. As an orchestration tool, Kubernetes can be used in conjunction with OpenWhisk, as well as IBM's Composer. As with the other three language runtime support here too, is quite large, ranging from Go over

¹⁵<https://kubeless.io/>

¹⁶<https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/>

¹⁷<https://github.com/kubeless/runtimes/tree/master/stable>

¹⁸<https://docs.fission.io/docs/concepts/components/core/controller/>

¹⁹<https://docs.fission.io/docs/languages/>

²⁰<https://www.openfaas.com/>

²¹<https://docs.openfaas.com/architecture/faas-provider/>

²²<https://docs.openfaas.com/architecture/autoscaling/>

Java and Python all the way to Nodejs²³.

In addition, four open source frameworks support three different types triggers [see figure ??], namely HTTP, Event and Schedule, natively. Moreover Fission, Kubeless and OpenFaaS are written in Go, OpenWhisk in Scala.

Concurrency test on Kubeless, OpenFaaS and Fission have shown that the latter framework with 2ms has the fastest response-time of all three. Moreover, its response times are consistent regardless of the amount of replicas present and the number of concurrent users [mohanty2018evaluation]. The test were conducted with 1, 25 and 50 replicas. Kubeless and OpenFaaS experience an exponential increase in response-time, with OpenFaaS showing a decrease in response time the more replicas are prevalent. Finally, Fission has a success ratio of 100% in all three tests, whereas Kubeless and OpenFaaS are facing a slight decrease from one to four percent.

Speaking of proprietary Function as a Service solution, the three market leaders are AWS, Azure and Google, respectively [kumar2019serverless]. Offering a vast amount of tooling and services, it can be challenging to choose one of these three, even though their model does not differ a lot. With all three providers, functions are billed on their number of incoming requests, the time a function runs to process a request, billed at a tenth of a second, and the memory allocated to a function. Due to the cloud, especially serverless, being such a volatile market, listing on pricing and memory size would presumably be obsolete in a few months, referencing to the providers' platforms. Instead, performance evaluations, concurrency and runtime will be used as an indicator for choosing a platform.

Currently on AWS the maximum amount of time a single function will last, is 15 minutes. After that time, the container the function is running in, will be killed, no matter whether it has finished processing. On Azure functions, by default 5 min are the maximum, which can be upgraded to 10 if needed. Google Cloud Functions comes last, with a standard execution time of one minute, which can be raised up to 9 minutes. Looking at the limits on concurrent instances, on AWS Lambda, the limit varies between 500 - 1000 depending on the region, although the burst concurrency limit can reach a maximum of 1000-3000 concurrent instances. These numbers only describe the initial limits, allowing to add each minute 500 additional instances. At this point it is important to mention that these limits refer to an account, not a function. 1000 might sound a lot in this context but depending on the amount of functions, 1000 concurrent running functions can be reached quite fast.

Regarding the execution time of AWS Lambda and Azure Functions the platforms not only differ in the execution time of cold and warm starts, but also when comparing two programming languages. As a compiled language, with its JVM, Java functions require more time to execute than an interpreter language like JavaScript. Tests have revealed, that there can be a significant gap receiving a result with a difference from 0.6 (JavaScript) and 1.7 (Java) seconds from cold to warm start on AWS and 9.8 (JavaScript) and 24.8 (Java) seconds on Azure [manner2018cold]. Moreover, Manner et al. have detected that the cold start time of Java compared to JavaScript is on each memory size on AWS by a factor of two to three higher. Azure with a factor of 2.5 shows equal latencies in cold starts. Due to different memory allocation approaches, explicit configuration on AWS and dynamic allocation on Azure,

²³<https://openwhisk.apache.org/>

the findings of latencies between the platform can not be compared to each other directly [manner2018cold].

Observed by other researchers as well, AWS appears to be very predictable in its memory size performance ratio [pawlik2019performance]. Even though the performance memory ratio is stable, the maximum CPU-performance increase seems to be reached with 2048MB Memory. 3008MB exhibits no significant improvement. IBM and Google experience similar ratios. In terms of delay, it is fewer the larger the memory size is, whereas Google's start delay is generally constant. On IBM though, it seems that resource consuming increases when a larger amount of resources is requested. [pawlik2019performance]. Function throughput, which is an indicator of concurrent processing, AWS reaches its maximum throughput most rapid, followed by IBM, Azure and Google, respectively. Whereas Google at the lower end increases throughput steadily, with a rise in calls, AWS stays sturdy at its maximum. IBM and Azure experience a small decrease over a long period of time. Lastly looking at the runtime overhead of AWS, IBM, Google and Azure, on AWS it is almost evenly low regardless of compiler or interpreter language. On Azure C# creates the least overhead, still being twice as much as AWS highest language runtime overhead, and Python the largest. On IBM the overhead is similarly high across the languages and Google has the least overhead with Node.js [pawlik2019performance].

Ein weiterer Punkt bei der Migration von Teilen einer bestehenden 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 genutzten proprietären FaaS-Lösungen [leitner2019mixed], 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.

[lopez2018comparison] hat diese drei Frameworks genauer untersucht und bei der sequentiell und parallele Planung sowie der Weitergabe des States der drei Orchestrierungs-Tools 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 [manner2018cold] und [jackson2018investigation]], 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 Funktion 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 [lopez2018comparison]. ADF und ASF sind wiederum in der Lage *Workflows* festzulegen, welche über Tage und Monate lauffähig sind.

Die Evaluierung des Overheads bei parallel geschalteten Funktionen 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 Ergebnisse legten zudem nahe, dass ASF zuverlässiger bei der Vorhersage des zu erwartenden 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 [lopez2018comparison]]. 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 ist²⁴. 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 [...]“²⁵. Die derzeitige Limitierung gleichzeitiger Funktionen in OpenWhisk liegt bei 100 pro *Namespace*²⁶.

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 5MB²⁷. ADF hingegen ermöglicht 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 hatten. Mit Payload betrug der Overhead in ms für Composer 298.4 und Step Functions 287.0, was eine Zunahme von 70% darstellt [siehe Abbildung ??]. Azure Durable Functions stieß bei diesem Test deutlich hervor. Mit einem Overhead von 766.2ms ohne Payload 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 [lopez2018comparison].

²⁴<https://github.com/apache/openwhisk-composer>

²⁵<https://github.com/apache/openwhisk-composer>

²⁶<https://github.com/apache/openwhisk/blob/master/docs/>

²⁷<https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-limits>

Abbildung 7: Overhead bei 5 sequentiellen Funktionen mit einem Payload von 32KB, nach [lopez2018comparison]

| Platform | Overhead (ms) | | Increase (%) |
|---------------------------|-----------------|--------------|--------------|
| | Without payload | With payload | |
| <i>IBM Composer</i> | 175.7 | 298.4 | 70% |
| <i>AWS Step Functions</i> | 168.0 | 287.0 | 71% |
| <i>Azure DF</i> | 766.2 | 859.5 | 12% |

Anhand der durch Lopez et al. gezeigten Ergebnisse lässt sich festhalten, dass AWS mit Step Function das ausgereifteste Orchestrierungs-Tool zur Verfügung stellt. Sowohl bei der sequentiellen als auch bei der parallelen Ausführung bietet 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 [lopez2018comparison]. 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*²⁸ bei parallelen Abläufen bietet zusätzlich eine etwas leichtere Umsetzung als AWS und IBM [lopez2018comparison].

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 sowie A/B Testen und Integrationstests. War OpenWhisk zu Beginn einer der ersten Anbieter, welcher lokale Unit-Tests unterstützte, so zogen AWS mit SAM²⁹ und Azure mit seinen sog. Function Core Tools³⁰ 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, bei dem Testen zu integrieren, so ist dies bei serverlosen Funktionen schwerer [roberts2017serverless]. 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 *Cold-Starts* können nicht akkurat wiedergegeben werden [racicot2019quality]. Mit der Abgabe der Hoheit über die Infrastruktur kann hinzukommend auch nicht mehr der Server bestimmt

²⁸<https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-cloud-backup>

²⁹https://aws.amazon.com/de/serverless/sam/?nc1=hl_s

³⁰<https://docs.microsoft.com/en-us/azure/azure-functions/functions-develop-local>

of serverless, mainly function as a service, exhibits a lack of monitoring, logging and debugging solutions [kritikos2018review]. Facing a lack of custom metrics, the operator solely relies on logging additional information needed, that are not provided by the platforms monitoring service. This is bound to the autonomy of the cloud provider for provisioning his platform. Enabling precise debugging on the environment would, as stated by [manner2019troubleshooting], take away the provider's control. Due to not having any control over the platform, it is crucial to monitor the functions constantly. In order to address this issue, Manner et al. provide a monitoring and debugging approach. Because of its comprehensiveness, it will be recommended to be used for setting up a monitoring and debugging solution for cloud vendor platforms. For convenience only, figure ?? shows the concept proposed by Manner et al. in a lightly modified way.

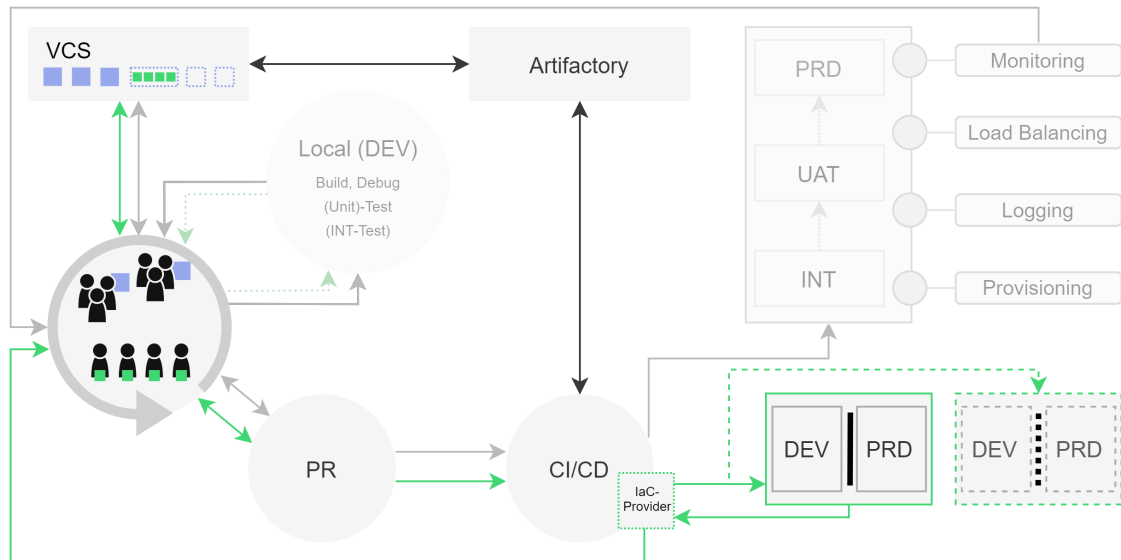
In the development-phase, the function should be implemented with at three parameters which are going to be logged to an external database or monitoring service. These parameters are the input, the context and the output of the function which is being called. The limitation to three parameters is chosen, to not have too many repercussions on the execution time of the function. At the same time, the the functions are providing enough data to reproduce failures in case they occur [manner2019troubleshooting]. After the functions have been tested on a development account, they will be forwarded to the production account. As a result of this, a clear distinction between the production- and the development account is not necessary since they are identical. In the second phase, the operational phase, the developer has to make use of monitoring service, which collects the data logged from the service. The received logging data will be further processed to determine a corridor for the execution times of the service. Whenever a service falls underneath a predefined execution time, an event will be triggered, which inform the ones in charge of operational purposes. With an alert beeing send, the troubleshooting-phase is launched. In order to create a test skeleton consisting of the three logged parameters, all functions failed or exceeding the corridor, are filtered to retrieve their logging-data. Afterward, the placeholders of the skeleton are replaced with the actual data to find the cause of the error [manner2019troubleshooting]. This process will ideally be combined with the development account, not to affect the performance of the production account. At last, it is important to asynchronously log the date before the function returns a response. After returning a response, the container will be discarded immediatly stopping any processing.

Speaking of monitoring, it is crucial to be aware of the modularity which comes along with Function as a Service. When monitoring an environment, login data from services and complementary parts will be collected. The data collected has to be stored with in a database, being able to later on run aggregation, sorting and other types of evaluation metrics upon. Implementing an open source FaaS framework, into an existing infrastructure, function monitoring can be included in the present monitoring solution. Due to being in charge of the infrastructure, metrics can be collected on a far deeper scale e.g. the container runtime itself and as well as from the underlying OS. In case the former landscape consists of several larger services, potential bottlenecks have to be eliminated. The bottlenecks could occur in form of connection pools to databases and need to be adjusted. With potentially many functions, that again have many instances, the connection pool has to scale dynamicall. Another solution is a central message queue all functions can provide their logging data to. As the message queue usually is faster than the database, it will collect the asynchronously logged data and forward it to the database.

When using a open source solution, Due to the granularity.

3.8 Summary

Abbildung 9: Modified DevOps pipeline



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

[al2019systematic] Serverless immer mehr bekanntheit und mehr Papers etc. ... Später market für funktionen die optimiert sind etc. [shafiei2020serverless]

Real time communication tool [shafiei2020serverless]

Real time tracking gps [shafiei2020serverless] da beide nicht auf dem Application state benötigen/ beruhen [shafiei2020serverless]

[hellerstein2018serverless]

5.2 Zusammenfassung

5.3 Weiterführende Forschung

Serverless Computing: A Survey of Opportunities, Challenges and Applications store for functions [shahrad2019architectural]