Bachelor Thesis

Degree in
Engineering Business Information Systems

obtaining the academic degree
Bachelor of Science (B.Sc.)

Topic

Construction and Validation of a Guideline for migrating a Microservice Architecture to Function as a Service

Submitted by

Gianni Pasqual

Born: 23.08.1997 Bad Soden am Taunus
Am Taunusblick 1, 65824 Schwalbach

Matriculation-Nr: 1179260
Wintersemester 2020
Faculty of Computer Science and Engineering
Frankfurt University of Applied Sciences

****

Examiner: Prof. Dr. Jürgen Jung
Second Examiner: Prof. Dr. Christian Baun

Handed in 18.03.2020

Abstract

Index of abbreviations

ADF	Azure Durable Functions
API	Application Programming Interface
ASF	Amazon Step Functions
AWS	Amazon Web Services
BaaS	Backend as a Service
BfF	Backends for Frontends
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
(D)DoS	(Distributed) Denial of Service
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 Progamming
OS	Operating System
PaaS	Platform as a Service
PoC	Proof of Concept
POM	Project Object Model
PR	Pull Request
PROD	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

Contents

1 Introduction

1.1 Objectives and Goals

The objective of the thesis is to develop a guideline for the migration of a microservice architecture to Function as a Service ("FaaS"), a so-called "serverless" Cloud-Computing concept. With the ever faster developing Cloud-Computing concept Function as a Service, another architectural concept has entered the technological landscape in the last four years. Till then, the prevailing architerctural pattern was the microservice architecture. Whether or not a company should align its IT infrasturcture to the emerging technology, the associated benefits and drawbacks are a decisive factor that needs to be taken into account. Furthermore, the question arises which parts of an existing infrastructure should be migrated and how migration affects operations and developent of a company.

This thesis will therefore examine the effects, not only on its advantages and disadvantages, but also on the IT landscape and organization. On the one hand, it is necessary to elaborate which changes are necessary with regard to team-internal and cross-team collaboration, as well as the general team composition and size. On the other hand, the question on having to adopt new frameworks arises. Besides, it has to be elaborated how many of the prevailent concepts can be adopted from the existing microservice architecture. Furthermore, it is crucial to consider how many and, above all, which components are to be migrated from the existing service landscape. It needs to be identified if there are any restrictions based on the programming language used.

Another central point is the question of implementation, since in addition to cloud vendors, there is also the possibility of using open-source projects to implement the desired concept. For this purpose, it must be evaluated which, in relation to the problem that should be solved, is the best alternative. course of action and whether, depending on the decision, there are tools that support processes such as deployment and general. At this point, the process of promotion was deliberately omitted, since the question of the continued existence of the usual test instances, given the possibilities of the later hosting solution, must first be answered. Once a decision has been made in favour of a vendor and thus a proprietary cloud solution, the possible vendor lock-in and the payment models should first be considered.

It should be worked out to what extent the approach of the micro-service architecture: "Do one thing, but do it good", can be acquired through the concept of "pay by use". A further focus is on the improvement or solution of problems in the existing microservice architecture. Specifically, it should be considered whether costs can be reduced, the development can be made easier and more efficient and whether the function-based development is ultimately sufficiently performant. The above-mentioned aspects of this new concept are to be treated systematically on the basis of a guideline in order to help IT and management managers with the migration.

1.2 Problem Statement

While microservice development has made application development more reactive, faster and more efficient, it has also increased the complexity of the development landscape. In addition to the actual microservices, a so-called core framework, which is integrated into all services, is required to ensure the implementation of basic functionalities. Besides, many additional tools were added to the process of development over time. These tools do, to some extend relief the developer of some tasks simultaneously add more complexity to development. Developers need to become expert in all additional tools and monitor them, to identify flaws and errors.

To touch on this topic only briefly, some tools will be introduced in the following. Swagger is used for code generation, Git for versioning, Jenkins for configuring the deployment pipelines, WildFly as the application server, Openshift for hosting applications, Geneos and AppDynamics for monitoring and many more. There is nothing wrong with this per se, but with complexity, the demands on the developer's increases. In order to build microservices, a whole ecosystem of tooling, monitoring and operational tasks is built around them. In addition, it is often not clear how large such a microservice should be. It should be large enough to handle a task autonomously, but the size always depends on the definition of the task. Furthermore, a service cannot be pushed directly into production, but must first run through various test instances before it can be used in production with other services.

These microservices are connected to each other via endpoints and can be scaled horizontally. If a service is called very frequently and exceeds the limit the container it is running in can handle, another instance of the service will be started in a short period. Because short in this sense is relative, there is, in many cases, a threshold which, when exceeded, will trigger the start of a new container. The incoming load of the requests is divided between the two services via a load balancer. The computing power of the services or the memory available to them is not increased, only the number of running services. With regard to scaling purposes, this sounds hugely efficient at first, but there is also a potential for optimization.

In practice, the application is broken down into smaller components, the microservices, which are hosted in a Kubernetes cluster on Pods. Those pods can be hosted internally on an on-premise solution or can be outsourced to a PaaS provider. A pod is the smallest deployable unit in a Kubernetes cluster and provides certain functionalities and programs to the service, see *3.1*. The capacity of a pod needs to be configured in advance, in terms of its computing power and memory. With the pod configured, once it is running, those capacities can not be scaled up or down anymore. In order to permanently be able to process a request, at least one instance of a microservice need to run all time. In practice, due to handling quickly appearing spikes, in many cases, at least two pods are active at any given moment.

If the service is receiving a continually high amount of traffic and is running IDLE only little, the concept of PaaS is maxed out and at its optimum. In practice, though, it is often the case that the containers are not entirely maxed out and the capacity that was paid for in adance gets wasted. If a container experiences 20% of its potential, 80% are running in IDLE, costing money.

Function as a Service tries to mitigate and even eliminate that effect, by enhancing the concept of *pay-per-user*, providing in build scalability and reducing operational costs.

1.3 Thesis Structure

The thesis is structured as follows. At first, the objectives and goals will be introduced and the underlying problem will be stated. In this section, the focus is on defining the expectations towards the guideline. Besides, aspects will be determined that should be highlighted during the process of migration. Next, the background will be covered, in order to provide a profound basis of the subject area Function as a Service is placed in. Therefore not only will the terms be defined and dissociated, but also benefits and drawbacks discussed. The purpose is to provide a basic understanding to follow better the guideline, which will be covered next.

The guideline at first gives an impression on the existing infrastructure to better identify with the purpose. Moreover, it can be used to determine whether the problem faced in the own company is similar to the one discussed in this thesis. Then, different steps during the process of migration will be examined and advice given on each subject which occurs. The guideline will end by summing up on the changes found, regarding the underlying microservice infrastructure and will be put in contrast to the picture drawn at the beginning of the guideline. Before Concluding on the topic, the guideline will be evaluated. During the process of evaluation, the guideline will be applied to the existing microservice landscape. A service will be depicted, according to aspects mentioned in the guideline, an be migrated to the platform of a proprietary cloud provider.

Lastly, in that section, adaptations and further research that was identified during the application will be summarized and stated accordingly. The thesis ends with a conclusion on the guideline and the implementation as well as on the general subject of Function as a service.

2 Background

Anfangs wurde damit begonnen die damals vorherrschenden Monolithen, die die Anwendungslandschaften vieler Konzerne dominierten und es zum Teil heute noch tun, in immer kleinere Teile zu zerlegen, um den enormen Wartungs- und Entwicklungsaufwand zu reduzieren. Mit steigender Attraktivität von Cloud Providern und den sich damit eröffnenden Möglichkeiten der Skalierung, wurden die Funktionalitäten der Monolithen in verschiedene kleinere Services unterteilt. Diese sogenannten Microservices sind unabhängig voneinander über Schnittstellen verbunden und kön-nen horizontal skaliert werden. Wird beispielsweise ein Service überdurchschnittlich frequentiert angesprochen, so wird in kurzer Zeit, wobei kurz in diesem Zusammenhang relativ ist, ein weiterer Microservice selber Funktionalität hochgefahren. Die eingehende Last der Anfragen wird über einen „Load-Balancer“ auf die beiden Services aufge-teilt, sodass diese nicht überlastet werden. Die Rechenleistung der Services bzw. der ihnen zur Verfügung stehende Speicher wird dabei nicht erhöht, lediglich die Anzahl der laufenden Services. Bezüglich Skalierungszwecken hört sich dies zunächst äußerst effizient an, jedoch besteht auch hier, genauso wie damals, durchaus Optimierungspotential. Mit Function as a Service sollen sollen Microservices in noch kleinere Module, den sog. Funktionen, heruntergebrochen werden. Ziel ist es, die Nachteile der Skalierung, des Provisionings, des Warten sowie des gesamten operationellen Aufwandes erheblich zu reduzieren. Um im Folgenden Leitfaden zur Migration einen fundierten überblick über das Themengebiet zu erhalten, soll zunächst das grundlegende Konzept von Function as a Service aufgearbeitet werden.

2.1 Definition Cloud Computing

Regarding a technology report, the term cloud computing can be traced back to the year 1996, where it initially has been used in a business plan from the company Compaq [**regalado2011coined**]. According to Regalado, several developers used the term cloud computing to discuss the future of internet business.

The basic concept of cloud computing is its dynamic, scalable, reliable and unlimited provisioning of resources via the internet. Over time, its concept has raised attention in literature but never became popular until 2006 [**fox2009above**]. In 2006 AWS released Elastic Compute Cloud (EC2), followed by Google with its App Engine in 2008. With those two products, the interest in cloud computing started to launch, and other companies like Heroku started to provide PaaS services as well. Even in that early state, Google introduces a stateful and a stateless layer into App Engine. The stateful layer was used to store and keep track of the actual state of the application, and the stateless layer was used to execute the applications inherent business logic [**fox2009above**]. In 2009 Berkeley released a study on the six most promising potentials of cloud computing that have been implemented by today different XaaS-Concepts, which are listed in the following:

1. The appearance of infinite computing resources on demand.
2. The elimination of an up-front commitment by cloud users.
3. The ability to pay for use of computing resources on a short-term basis as needed.
4. Economies of scale that significantly reduced cost due to many, very large data centers.
5. Simplifying operation and increasing utilization via resource virtualization.
6. Higher hardware utilization by multiplexing workloads from different organizations.

After 16 previous definitions, the National Institute of Standards and Technology (NIST) published a final definition on cloud computing in 2011, that is referred to in several papers [**mell2011nist**]. The definition, according to NIST, describes cloud computing as a model consisting of five essential characteristics, three service models - IaaS, PaaS and SaaS, respectively - as well as four deployment models.

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

Roughly speaking, the five essential characteristics are, the ability to automatically provision resources based on present demands (*on-demand self-service*), the ability to access them from all common devices (*broad network access*), the ability to distribute resources as efficient as possible to those consumers which currently need them, whereas the exact physical position of these resources do not matter to the consumer (*resource pooling*), the ability to automatically scale resources, based on incoming traffic (*rapid elasticity*) and the possibility to monitor and limit resource utilization, which has to be transparent to the customer (*measured service*).

With the service models, only IaaS, PaaS and SaaS are distinguished. Other XaaS-solutions like FaaS and BaaS have not been invented at that time, but they do follow the described pattern. According to NIST, four different deployment models exist, that can be used to provide the application code to the cloud provider. Using a *private cloud* approach, it is the companies responsibility to provision its infrastructure. The infrastructure itself does not necessarily need to be hosted on-premise and can be obtained from a third party PaaS or IaaS provider. With the deployment to a *community cloud*, several companies share the same cloud, that is either operated jointly by them or provided by a third party. Second, to the last, there is the possibility to use the *public cloud*, in which the company uses the facilities and resources of a third-party provider. The last deployment model is the so-called *hybird cloud* approach, which is a combination of several aspects from the previous three [**mell2011nist**].

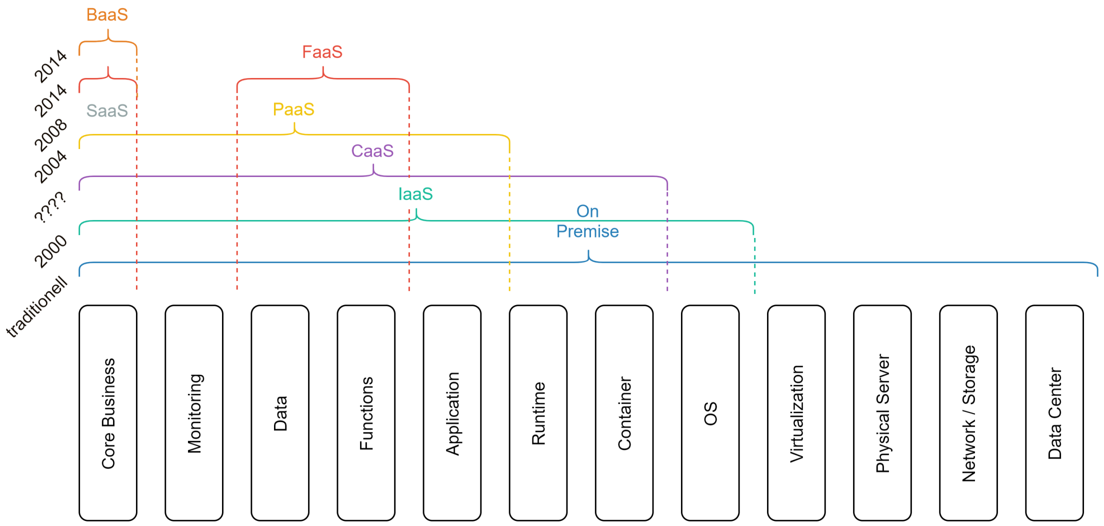


Figure 1: Overview of Cloud Computing Service-Models

Besides the three mentioned cloud computing models depicted by NIST, and the marked BaaS and FaaS solutions, others exist as well, that are depicted in Figure **??**. The displayed cloud concepts have evolved over time from 2011 until today. Figure **??** provides a gross overview of their level of operation and the year they have been introduced.

2.2 Function as a Service

Function as a Service is a so-called *serverless* cloud computing concept, which initially was introduced by AWS Lambda in 2014. In 2014 Lambda was only available as a preview release and launched for commercial usage in 2015 [1]. Almost one year later, Microsoft, Google and IBM entered the market as well, with Azure Functions, Google Cloud Functions and OpenWhisk, respectively. Regarding IaaS and PaaS, Function as a Service further abstracts operational tasks, such as provisioning of resources, monitoring, scaling, failure recovery and availability, away from the developer toward the platform. The market around Function as a Service has evolved steadily and meanwhile offers a vast number of open-source and proprietary cloud providers. A detailed overview of the existing solutions is provided by the CNCF[2]. Moreover, CNCF provides an overview of tools, which is dealt with in a later section.

Due to FaaS being a relatively new technology, there is no official documentation or reference architecture as i.e. provided for Cloud Computing by NIST. The consequence is a lack of the underlying architecture and different implementation of the providers, resulting in inconsistency. The lack of uniform implementation over the existing platforms exacerbates multi-cloud solutions across them [**mell2011nist**].
Nonetheless, there is consensus on the definition and functionality of FaaS across the providers, for the most part. Microsoft i.e. defines Azure Azure Functions as an „*[...] 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]. With that definition, Microsoft does not describe the operational benefits of FaaS but its technical capabilities. The concept should be used to define certain events that are activated by the occurrence of predefied trigger, which could be a HTTP-call, and bindings to a database or another service. Looking at the definition of AWS Lambda, the same technical benefits are described: „*AWS Lambda [...] runs your code in response to events [,] automatically manages the underlying compute resources [...] [and] extend[s] other AWS services with custom logic [...]*“ but also the operational benefits are taken into account „*[...][Lambda] performs all the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, code and security patch deployment, and code monitoring and logging. All you need to do is supply the code.*“[4].

[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/

[4] https://aws.amazon.com/lambda/

Besides the basic event-based HTTP-triggers and scheduled Function invocations, most solutions, both open-source and proprietary, provide one more trigger that strongly depends on the underlying platform. This trigger, as mentioned by Microsoft, is the binding to other services supplied by the platform. Events of a function are connected to the different BaaS solutions of the platform, further discussed in*Dissociation of FaaS and Serverless*. They are triggered when something in a database changes, a new user was registered or a Push-Notification -like AWS SSN or Google Pub/Sub etc- needs to be sent.

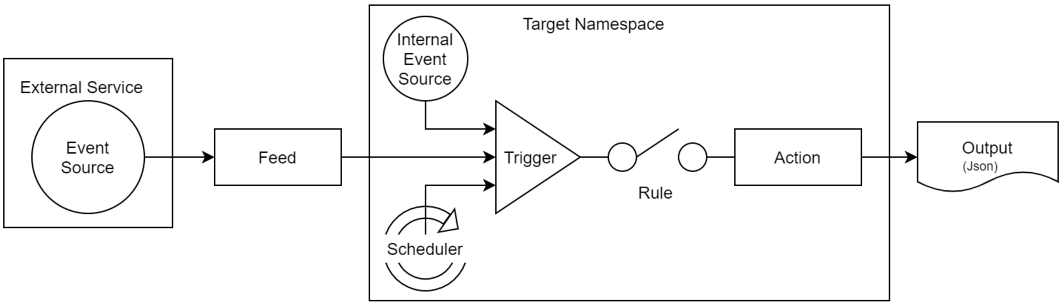


Figure 2: [5]

Figure **??**, based on OpenWhisk, provides an overview of the different types of event-triggers that are currently used by the three largest FaaS-providers, namely AWS, Azure and OpenWhisk. Due to OpenWhisk, formerly being backed by a large company (IBM), in literature, its model is often used as a reference for the other FaaS implementations of the other cloud vendors. It can be assumed that the architectural concept of FaaS at the other providers is not that different from OpenWhisk, because it is supplied in a large cloud environment (IBM) [**van2019spec**]. Currently, OpenWhisk is developed as an open-source framework, backed by Apache.

Another fundamental concept of FaaS, which is in accordance with the fifth essential characteristic of the NIST definition on cloud computing, the *measured service*, is its scalability by the so-called pay-per-use model. It ensures that the customer only pays for the resources and computing capacity he used and not for the ones reserved, i.e. to cope with possible increases in traffic.

What can not be seen on the picture, but is essential to FaaS, is its statelessness, as further discussed in *Benefits and Drawbacks*. Even though the container the functions runs in do have an application state inherent to them, utilizing that state is not an option. Functions are only active for a certain period and automatically will be discarded when reaching the limit of the platform. Therefore a request hitting a *warm* function technically can access its state and do a calculation on it, but it is uncertain by which container an incoming request is served.

Another important point, which can be found with other providers as well, is registered in the definition of OpenWhisk. On the one hand developers „*[...] can focus on building amazing and efficient applications [...]*“, the other hand „*[...] developers [are writing] functional logic [...] in any supported programming language, that can be dynamically scheduled [...]*“[6]. The former account the abstraction of servers, infrastructure and maintenance, at least to some extent. For the developer, it is not necessary to have any knowledge about the functioning of the underlying OS or the like, to provide working code snippets or an entire application. To be put in exaggeration, the developer cannot even configure the platform, whereas the only thing left, is to rely on the code he writes. By abstracting most of the provisioning tasks to the platform, the developer can only control aspects the provider allows him to control. The latter aims on a variety of programming languages. This is not solely a benefit of Function as a Service and also applies to other architectures like microservices. Nevertheless, it provides the developer with more flexibility addressing a wider group of potential users.

Figure **??** depicts the general functioning of Function as a Service in a broader context. On the top right, the function-pool can be seen. When the developer uploads functions to the platform for later execution, the function will be stored in the function pool until it gets triggered. As soon as a trigger activates the function, a copy of the function will be instantiated. Before the actual business logic of the function can be processed, necessary modules and libraries, as well as other dependencies, need to be loaded. Depending on the frequency a function is triggered by clients, the process of instantiating new function copies, based on the function in the function-pool, will be repeated.

[6] http://OpenWhisk.apache.org/

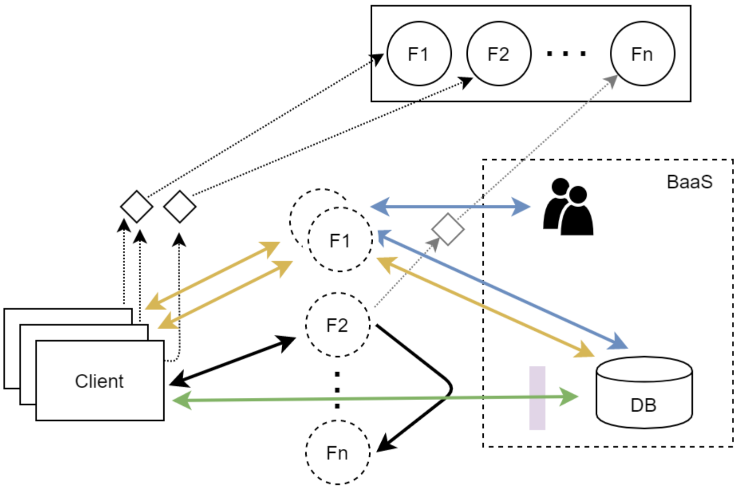


Figure 3: Anwendung mit FaaS und (P)BaaS, angelehnt an [**shafiei2020serverless**]

The function being called can either process the incoming request directly and return the result to the client or can access underlying BaaS services during the process of processing, as seen with F1. The services accessed and activated by a function can range from simple database queries up to register a new user, saving his profile data and provide certain permission to. The possibilities are only limited by the service ecosystem of the respective provider. Just like calling another function, the developer can also chain several functions to return the desired output. Chaining is only limited by the provider's orchestration solution and the maximum number of functions.

To better and more uniformly define FaaS and thus create a cross-platform standard, there is a growing demand in industry and research for a reference architecture such as that available for cloud or grid computing [**liu2011nist**], [**foster2003grid**]. The absence of such an architecture hinders the establishment of best practices, design patterns and a more precise an overview of how the field of Function as a Service is developing [**leitner2019mixed**]. Initial suggestions for what the reference architecture of FaaS might look like have already emerged. Reason to that is increasing popularity of FaaS in research and literature over the years, starting in 2016 [**van2019spec**]. While the topic was not addressed until 2016, the presence of FaaS has been increasing continuously from then until 2019, in journals, conferences and workshops [**Yussupov2019˙SystematicMappingStudyFaaS**].

2.3 Dissociation of FaaS and Serverless

Serverless computing, often referred to as serverless, is a sub-discipline of cloud computing that has evolved from the virtualization of computing power, storage and networks [**jackson2018investigation**]. As often, it is not easy to differentiate or distinguish between newly developing technologies. Initially, serverless stood for applications which partially or completely relied on third-party providers, on so-called Backends as a Service (BaaS) [see Figure **??**]. To perform server-side tasks such as database queries, user administration, Push-notification and other typically used web- and app-functionalities, the different vendors provide a vast service-ecosystem to fulfil those needs [**fowler2018serverless**].

With FaaS, the serverless concept has been extended so that the server-side logic is no longer completely provided by a third party. With FaaS, the developer has the possibility to execute business logic as a middleware between the client and the backend. This logic can be implemented by the developer himself. Serverless is one of the most frequently used buzzwords in IT in the last few years, whereas the term „serverless“ suggests something different than what it actually is. The term implies the absence of servers, which is true from a physical perspective but not in regard to the responsibilities. In the broadest sense, serverless means a shift of responsibilities and operational tasks. Developers of a serverless application, do not have to deal with the operational activities such as provisioning, monitoring, maintenance, scalability and the robustness of the infrastructure [**baldini2017serverless**]. However, the transfer of responsibilities is only possible with a certain degree of vendor lock-in, whereby the providers ensure that with FaaS, many services, provided by their ecosystem are used in conjunction [**kritikos2018review**]. As many triggers as possible, presented in the previous section, should be used when building serverless applications.

Function as a Service is thus largely derived from Event-Driven Computing, which is used primarily in UI development and has been adapted by serverless computing. Nevertheless, the term serverless is equated with FaaS in many cases, loudly in 58% of the cases studies by Leitner et al., which makes it difficult to distinguish between the two [**leitner2019mixed**]. Figure **??** illustrates the relationship between the different concepts.

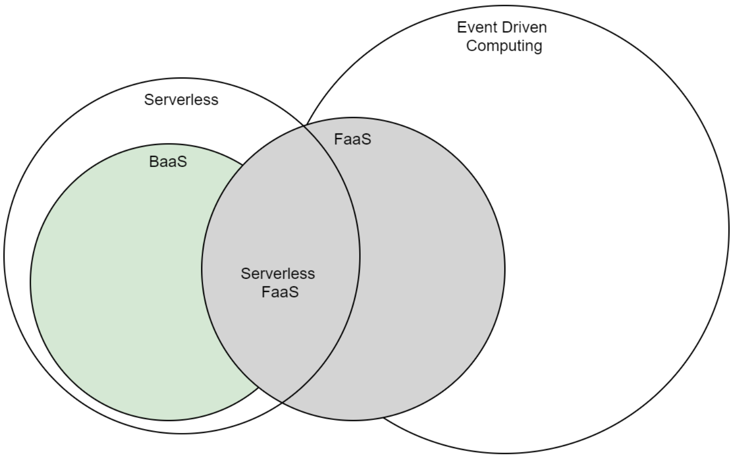


Figure 4: Serverless Concept, including FaaS and BaaS

If an application subsequently consists only of the two previously mentioned serverless components, FaaS and BaaS, a distinction is often made between a *pure serverless* application, i.e. an application whose operational part has been completely outsourced to a cloud provider, and a *hybrid serverless* application [**leitner2019mixed**]. In the latter case, the functions often act as glue. Tasks that run very frequently and most of their time are running IDLE, do not claim any CPU or memory and as a consequence are not charged for. In IaaS and PaaS, these tasks probably run in a separate container, and therefore permanently are running.

It has to be mentioned that a large part of the infrastructure-related tasks is also performed by the cloud provider when using PaaS. However, FaaS goes a step further at this point. With Paas, pre-built packages of an application are provided on the runtime of the corresponding platform, so the developers still have to take care of the application structure [**kaplan2019framework**]. In FaaS, however, those tasks are performed by the platform provider and the developer can concentrate solely on the business logic and the respective event triggers.

Finally, the frequently used term „NoOps“ [**fowler2018serverless**] should be addressed, that mistakenly implies the absence of operational tasks. Although a large part of the maintenance is outsourced, this is not to be equated with the assumption that the operations of DevOps will be discontinued, as is suggested [**fowler2018serverless**]. Though the cloud-vendor handles load balancing, automatic scaling, security aspects and software patches, other tasks still need to be addressed by the developers and operators. It is still up to the developers to write high-quality code and test it locally as well as in the cloud environment to ensure its performance. The complexity is, therefore, not completely eliminated but shifted to a certain extent [**eivy2017wary**]. Looking at AWS, a large number of configurations must be made, to properly and securely operate in production. Starting with configuring the AWS API-Gateway, for being able to shift traffic for, i.e. testing purposes, the IAM needs to be configured afterwards. Furthermore, there is not one IAM to be configured but several, depending on the amount and type of functions as well as other services used on AWS. Although existing configuration, once made, can later be applied to other functions, it is essential to carefully configure them, to not run into security problems in a frequently used multitenancy environment.

2.4 Benefits and Drawbacks

The potential and challenges listed below, which literature and users of FaaS face, are partly due to the concept itself, but also partly to the implementation of the respective suppliers or open source solutions. The challenges are therefore first explained and, if already addressed in the literature, suggested solutions are listed. For both, potentials and challenges, reference is made to FaaS as well as to BaaS, since these two categories are used in combination in most cases and correlate strongly in terms of architectural conditions [see Figure **??** and „statelessness section *Function as a Service*].

Cost efficiency
A very frequently mentioned benefit of serverless computing and especially FaaS, is its cost efficiency [**lee2018evaluation**]. By only paying for compute time and memory size allocated to a function, the amount of functions store on the platform do not matter to the developer. In addition, the event-driven nature of FaaS and the remote runtime provisioning, solely when an event triggers the function, are critical for cost efficiency [**feng2018exploring**]. Even though the cost efficiency of FaaS can be a huge benefit for some use cases, FaaS is not an ideal solution for all use cases. When functions run over a long period and get triggered very frequently, PaaS solutions are likely the better choice. The price for executing a function that runs for one minute, is generally lower when executing the same code on a PaaS server with similar resources [**jonas2019cloud**]. Looking at PaaS, the primary purpose is to execute functions that experience permanently high traffic. Functions instead are primarily intended to solve simple, high-traffic tasks where the user only has to pay for the resources that the function uses when it is called. Pikes do not need to be recorded to adapt the memory or computing size of

an application running in a PaaS hosted container. No adjustments need to be made based on historical data. Nevertheless, Elgama et al. exhibit that also with Function as a Service the need to decomposite functional components of an application is still required, even in regard to the pay-per-use model. Not separating functionalities effectively into separate FaaS functions with different memory sizes can have significant influences on costs [**elgamal2018costless**].

Time to Market / Lead Time Development

Tough lead-time development is not the primary benefit of using FaaS. It is mentioned in several papers as being one of the key benefit serverless enables [**adzic2017serverless**] [**ast2017self**]. According to Leitner et al. Time to Market is not one of FaaS primary advantages, which is according to the study, the scalability and elasticity [**leitner2019mixed**]. With FaaS, it is possible to deploy code within seconds and minutes. In regard to different testing approaches, such as canary releases, many cloud vendors, such as AWS, provide simple predefined solutions for such purposes. By utilizing services such as API gateways, the platform supports fast development and deployment. Changes can be quickly made to functions and tested individually in the cloud [**sewak2018winning**].

Physical Location

Another point to consider is the physical location of the functions. It is the provider's responsibility to use its resources as efficiently as possible. Therefore, the provider decides on which node in his infrastructure a function should be executed. If the provider subsequently locates functions that are highly data-dependent physically far apart from each other, effects on performance will be more liekly faced by the end-users of the application [**shafiei2020serverless**]. Although many providers, such as AWS, Microsoft or Google, already offer the option of defining the region or a so-called cluster, this only allows the location of the function to be limited to a certain area, but not to exactly define the server the functions should be executed on. FaaS cannot match the performance of serverfull applications [**shafiei2020serverless**].

Serverless

At this point, it might seem relatively trivial, but still, a disadvantage compared to solutions like PaaS or IaaS. We are talking about the loss of server-side optimization. As the name already implies, servers do not exist for users or, in the case of a software solution based on BaaS, the developer has no control over the backend. He can only adapt the databases, object storage or authentication to the structure of his data and implement access restrictions via rules. The services are predefined by the respective vendor and cannot be adapted to the corresponding client types, such as a tablet, mobile phone or desktop. They can not be optimized to meet the requirements of the „Backends For Frontends“ pattern, written by Sam Newman[7]. The topic of only one BfF-service for multiple clients is further discussed by Perera et al. in [**perera2018rule**]. There is only one version of each type (databases, object storage, authentication, and so on) to which developers are restricted. Any user-defined logic must, therefore, be outsourced to the client, as it cannot be implemented on the backend side [**fowler2018serverless**]. With FaaS, this effect can be mitigated by implementing lightweight logic in functions that act as middleware on the server-side. This involves interacting with the various BaaS services, using the triggers provided by the cloud vendorse, as explained in section *Function as a Service*.

Statelessness

The statelessness of functions, on the one hand, enables the pay-per-use model and scale-to-zero functionality of FaaS, but on the other hand, presents a problems practitioners have to deal with. The core of applications is to solve tasks that require many different steps which need to be connected in a meaningful way, in order to provide the desired output. Thus, even with the function-based structure of applications, it is essential that the functions can intercommunicate and query the program status from each other without inconsistencies. Due to the short runtime of functions, the exchange of information must take place at a corresponding speed. However, fast and accurate state sharing still proves to be problematic when considering the speed of serverless applications compared to serverfull applications, as pointed out in a report from Berkeley [**jonas2019cloud**].

To persist the state of an application, the Key-Value-Storages and object storages offered by the providers can be used. Object storages of the different providers, such as AWS S3, Google Cloud Storage, Azure Blob Storage, are not very expensive in terms of storing data, whereas the costs for accessing the storage and the latency of up to 20ms are high [**jonas2019cloud**]. The key-value storages are the better choice in this case, as their response times are lower with 8 - 15ms, but they are significantly more expensive than object storages, in terms of their input/output operations per second [**jonas2019cloud**].

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

Taking the necessity of fast exchange of information between functions and external storage mediums into account, it might be more appropriate to pass a functions state, when interconnecting functions, from one function to another without querying the current state from an external database. As mentioned above at the *physical location* of functions, it is likely that functions are not started on the same node. The load balancer of the respective provider decides on which node a copy of the function is executed, depending on the current load on the infrastructure. Even though in a perfect world, the transfer of the application state is faster when interconnecting functions than querying a database, the advantage is mitigated by factors inherent to FaaS. Those factors are not inherent to the concept itself but to the technical implementation.
As a consequence, the effect of fast state transfer between functions is relativized by the functions startup latencies and physical distance from each other, which can not be influenced by the developer. To counteract this, Shafiei et al. suggest marking interrelated functions to draw a dependency graph and starting possible subsequent functions directly at the initial call of the first function [**shafiei2020serverless**].

If the user were allowed to determine on which instance (node) a function should run, the concept of FaaS would be undermined (see *Optimal Utilization of Data Centers*). Enabling developers to precisely choose the instance a function-pool should run on capacity would need to be reserved for those users. The provider would have to hold the capacity back, in order the functions, specific to a user, are called. The vendor would no longer be in charge of its platform and could not free unused capacity to other users in order to increase efficiency [see [**fowler2018serverless**]].

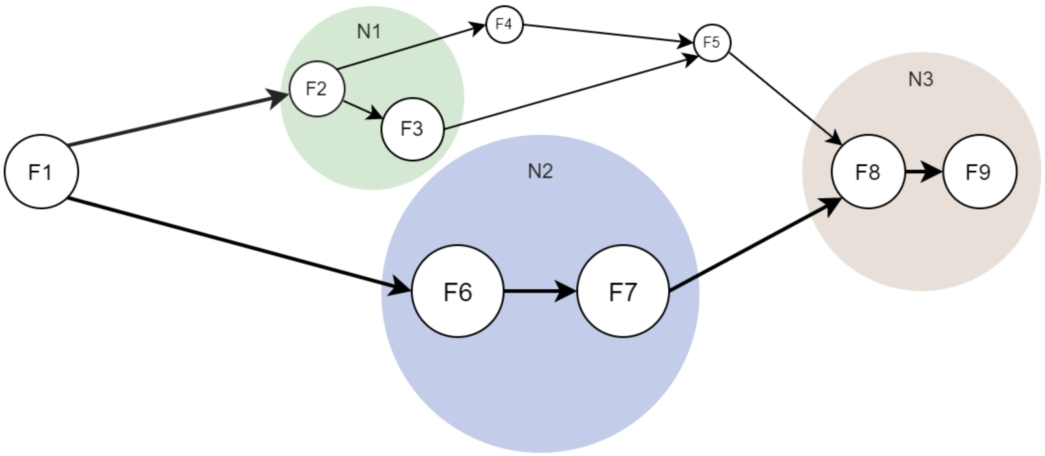


Figure 5: Dependency Graph nach [**shafiei2020serverless**]

If each user on determined ch node, i.e. Node-X, his functions should run on, in order to pass on the program status with as little physical latency as possible, an optimal workload could no longer be guaranteed. The provider would always have to retain capacity from Node-X and could not release it for other functions. He would always have to retain some part of the capacity of Node-X in case inactive functions are allocated. A dependency graph, as shown in the Figure **??** would bypass the need of determining a specific node. Specifying a node would no longer be necessary, since the provider could start functions, connected via the dependency graph, simultaneously. With the time of cold starts and dependency instantiation saved, the only the physical location of the functions could affect latencies. The provider's load balancer could be able to start functions as usual. Since latencies when starting a service are considerably higher than those caused by the location, their elimination would considerably accelerate the state propagation [**aditya2019will**] [**jackson2018investigation**]. In addition, the language used for the function will influence its startup latency as well. Therefore object-oriented and functional programming languages should be weighed against each other [**manner2018cold**].

By combining this approach, which already accelerates intercommunication, with a simple neuronal network, predictions on functions could be made. Looking at Figure **??**, by weighting the edges connecting the nodes, the probability of hitting F2 instead of F6 could be further determined. Thereby the platform could use their resources even more effectively because not all functions connected in the dependency graph would have to be started. However, if the prediction is in favour of F2 but F6 is triggered, then the latency, caused by its cold start, could increase, but also changes to the edges of the graph would be made.

Multitenancy

Although the concept of FaaS is based on the fact that the provider controls and maintains a variety of functions from different customers, each customer should feel as if they are the only one. However, this poses additional challenges on the platform operators, as they must reliably guarantee resource isolation and access restrictions. Even though the providers manage to guarantee resource isolation reliably, the user still has to set up his security rules properly.

Conditioned by the nature of serverless applications, they are based on the communication and interconnection of services and functions. For each function, a security policy needs to be set up. If the policy is not configured correctly, this already represents a security risk. The functions and services could grant access to data, which was not intended by the user [8][9].

Although providers like AWS should be experienced enough by now not to expect such problems anymore, caution is still required [**fowler2018serverless**]. They constantly have to deal with security, robustness and performance of their infrastructure. No customer should see another's data, no error should compromise the stability of other functions, and no sudden spike from one customer should affect the performance of another's customers functions. The interaction between RPCs and container security must be permanent and be ensured by careful security management [**mcgrath2017serverless**].

Cold-Starts

Cold starts refer to starting a container in which a copy of the required functions is provided. If a function has not been used for a long time or is being addressed for the first time, everything needs to be prepared for execution at first. Especially with concise functions, they pose a considerable problem, since their startup time can be up to ten times the actual execution time [**shahrad2019architectural**]. A container must be started on a node and the function and its dependencies have to be loaded. The time it takes for the instance to be ready for processing a request depends on various factors. Factors can be the programming language, the required libraries, the size of the function (amount of code) and the hardware on which the function is instantiated

[8] https://awsinsider.net/articles/2017/06/20/voter-data-leak.aspx

[9] https://searchsecurity.techtarget.com/news/450422962/Another-AWS-cloud-data-leakage-due-to-misconfiguration

[**shafiei2020serverless**] [**jonas2019cloud**]. If a function is called very frequently, the platform will keep these functions „warm“ for a certain time (up to one hour for AWS [**roberts2017serverless**]), so that the probability of a new cold start tends towards zero. In contrast, functions that are called only once per hour are more likely to experience the effects of cold starts, each time they are accessed [**roberts2017serverless**].

To counteract this, Castro et al. propose to keep a kind of stem cell container. The container would hold already instantiated but empty containers for the supported programming languages [**castro2019server**]. The containers are not customer-specific and can, therefore, be used generically. This would eliminate two of the above-mentioned issues, the programming language and the hardware, which influence the startup time of the functions negatively. Only the business code of the functions and their dependencies need to be loaded. Ishakian et al. have also shown that by increasing the memory capacity the time until a function is ready to process a request can be reduced [**ishakian2018serving**]. If this is combined with the stem cell approach, the loading of the program code and its libraries will be accelerated. Of course, the additional memory does not come for free, but in this case, it is up to the company to weigh the additional costs against the performance increases.

3 Guideline 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. Afterwards, criteria for selecting the right service will be stated. Those criteria 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, OpenFaaS or Kubeless dedicate greater control to 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 will be investigated. Changes in an organizational structure regarding the development team, effects on the CI/CD pipeline, challenges faced with local testing and the VCS, will be addressed. At this point, it should be outlined, that this thesis differentiates between the testing that is done locally during development and testing, which needs to be done on the ecosystem of the platform itself. With the development process covered, the guideline on migration will end with the operational part. 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. Ending with a summary on the constructed guideline, changes affecting the current infrastructure will be outlined.

3.1 Describing the existing architecture

In the follwing, the current process of development and operations is described, to later on better illustrate the effects of introducing Function as a Service to the existing microservice architecture, see [**newman2015building**]. Figure **??** provides a high-level overview of the prevalent DevOps-Pipeline. In the final stage of this section, by summing up on the construction of the guideline, Figure **??** will be examined regarding necessary changes.

Starting with the development team, it consists of around eleven developers that are working after Scrum, an agile process framework,see [**pichler2013scrum**]. During a sprint, the team splits up in groups of two to four, to work on the tasks they committed on. Thereby the primary language used for developing the services is Java. Following a typical pattern, the basic structure of a service is defined in a YAML-file [10]. In this file configurations regarding the endpoints of a service, its request and response parameters and other basic configuration can be made. The YAML-file then gets passed to swagger-codegen, which generates a service skeleton with the according interfaces. Using Maven as an automation tool for service development, when building the service, all required dependencies, specified in the POM, will be loaded. The POM itself is an XML-file, including information about the service. The file documents target locations, dependencies, versions and other default configurations for building the service. After the initial build of the service, the business logic can be implemented. To keep track of different versions of services as well as the entire service landscape of the platform, a VCS like Github, Bitbucket or Tortoise can be used. While working on the business logic, the service will be built and tested several times. The tests are executed in the local development environment, using mock datafromtest-databases. By the time the service is documented, a PR can be created. To guarantee a certain service-quality, all services embrace parts of a so-called core-service. The core-service enforces certain standard functionalities in terms of logging, messaging, caching, circuit breakers, standard exceptions and many other tasks, other services have to incorporate. Given that the service implements the standard correctly and the code-review reveals no flaws, by approving the PR, the CI/CD pipeline is triggered.

[10] https://yaml.org/

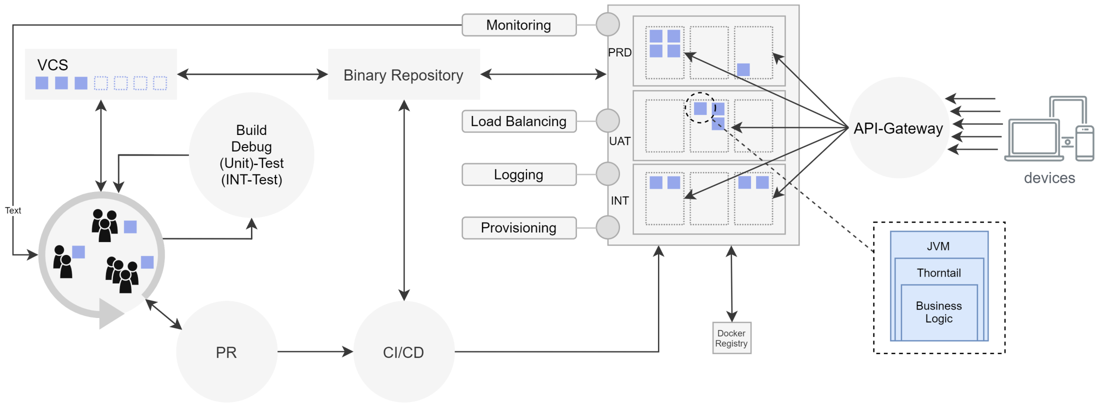


Figure 6: DevOps Pipeline

Looking at the CI/CD pipeline, different products can be used for automation. In the financial sector, two popular choices are Jenkins and TeamCity. The purpose of the pipeline is to build a service, execute tests, and provide the services war-files and jar-files to a binary repository. The binary repository itself is connected with the VCS as well as with the CI/CD pipeline. By triggering a build process for one of the services its binaries as well as complementary program parts – such as Git, the JVM, Thorntail etc. –, will be put into a Dockerfile. Based on the Dockerfile, a baseimage will be built, containing all specified programs that are needed for the program to run correctly. This baseimage will be forwarded to a container where the application runs on and is stored in the Docker registry for later usage. When a new instance of a service is needed, it can be loaded directly from the Docker-registry. To ease working with containers, Kubernetes was installed, which is an open-source framework for automating deployments, scaling and management of containerized applications.

Microservices are usually hosted on-premise, PaaS or IaaS solutions and run in a container. A pre-defined amount of memory and computing resources are allocated to a container, enabling it to handle a certain magnitude of incoming traffic. When the traffic exceeds a predetermined threshold, another instance of the service will be provided on the platform, using the baseimage from the docker registry. Due to the horizontal scaling nature of containers, each instance has a fixed amount of resources which are constructed to deal with incoming traffic. In practice, the containers are running most of the time in IDLE. Even though microservices already have decoupled many monolithic applications into smaller, more manageable systems, they still incorporate different functionalities. In reality, microservices can get quite large and therefore need to be grouped by bounded context to better suit the concept of FaaS. Because of their size, it takes time to launch a new container, in case the incoming traffic rapidly increases. Therefore it is good practice to provision at least two service instances if historical data has shown frequent bursts in load.

3.2 Scrutinising the Decision of Migration

Whereas migrating from a monolithic application to Function as a Service seems far more challenging, due to the assumed size and complexity of the monolith, migrating from a microservice architecture to FaaS can be challenging as well. With a monolithic application, partitions need to be identified in the first place and afterwards can be broken down into small services. 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, issues that have to be dealt with will remain. When migrating, a solution for outsourcing the internal state of the microservice has to be taken into account. Furthermore, dependencies need to be reduced to a minimum and the code probably has to be optimized to better suit the concept of FaaS.

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. When a system is already composed of many small services, the underlying infrastructure likely consists of VMs or containers. Those VMs and containers often run in a PaaS environments and it is not possible to scale them to *zero*. Scaling to zero means to not have any containers running at all, when no requests are forwarded to them for a specified period. This way, unused computing capacity can be released, in favour of other services to consume it. Moreover, with Faas, the expense of provisioning the service-landscape, managing the underlying infrastructure and coping with operational tasks, such as monitoring, load balancing are being outsourced to the respective Cloud Provider. On one hand Function as a Service can help to achive the describe state. On the other hand that state does not come without restrictions, which need to be considered. The restrictions that come along with the ease of development have to be traded off against its benefits.

To benefit as much as possible by 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 to a minimum, due to their effect on start-up times [**manner2018cold**]. Especially with Python, Nodejs, and Java [**puripunpinyo2017effect**], loading all dependencies required can have an impact on cold starts. Also, the dependencies might again interconnect with other dependencies, resulting in a considerable number of dependencies, that should not be underestimated. With FaaS, each unnecessary dependency will increase the start-up time of every single Function being started.

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. Being precise, there is a persistent state in each container of a function, but whether an incoming request will hit a particular 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 maximum amount of time, of a frequently called function until it gets killed, is 45 minutes[11]. 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 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 vast amount of traffic, once running on a FaaS platform and once running in a docker container on PaaS, will be more expensive implemented with FaaS, than it will be with PaaS [**jonas2019cloud**]. 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 of the system 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

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

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, provisioning, versioning and providing monitoring solutions. On the other side, new complexity will appear in other areas. To cope with the complexity given by a potentially large number of functions, the issue of mono-repo and poly-repo arises, as discussed in a later section.

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 language support for the language used in the application or service. If the platform does not provide the required language, to avoid the hassle of finding a workaround, an alternative solution should be chosen. 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, parts of 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 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 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 vendor should 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. 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. Moreover, if an application requires user authentication, different types of databases, messaging services and hosting, each of the platforms 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 services.

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 rely on [**manner2019troubleshooting**]. To reproduce the error, the custom logging data can be crucial to find the cause of the problem 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 have to 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, but increase operational tasks.

Depending on how one looks at it, Function as a Service in conjunction with Backend as a Service can be depicted as an advantage or disadvantage. Even though the two concepts undertake many tasks, for many tasks, additional services need to be consulted. I.e. when building a serverless application, for messaging an SNS-service is needed, for storing the state a database-service is needed, for registring new users an authentication-service and for monitoring a monitoring-service needs to be included. Not yet mentioned CI/CD integration, API-Gateway and many more.

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, an open-source provider needs to be chosen. If the present application is built upon Kubernetes, for the purpose of virtualization, Kubeless, OpenWhisk, OpenFaaS 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 underlying platform. Introducing FaaS to an upfront ecosystem will increase its performance and efficiency by freeing unused resources. At the same time, an operations team will be needed that is in charge of implementing a monitoring solution and configuring the properties of the FaaS framework [**mohanty2018evaluation**].

Furthermore, the team needs to acquire new skill sets to configure security policies correctly, interconnect function with each other, define different triggers on functions, deal with concurrency setting [12], and many more aspects add complexity to new fields. If the company aims at reducing infrastructural complexities, then Function as a Service can provide support on 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.

3.3.1 Open-Source Solutions

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

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 any files, objects, binary data and other formats in. Since it is reusing everything from Kubernetes, Kubeless solely relies on the API integration. To scale functions, Kubeless makes use of the Horizontal-Pod-Autoscaler included in Kubernetes. Depending on the runtime, metric data can be monitored with Prometheus.
Regarding the deployment of new Functions, the API, ConfigMaps and Ingress are utilised[13]. Thus being built on top of Kubernetes, making use of its CRDs[14] 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 on supported runtimes can be found on *github*[15]. 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

[12] https://docs.aws.amazon.com/lambda/latest/dg/configuration-concurrency.html
[13] https://kubeless.io/
[14] https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/
[15] https://github.com/kubeless/runtimes/tree/master/stable

function with a CPU request limit.

Fission is a serverless framework build on top of an underlying Kubernetes infrastructure. Just like Kubeless, Fission does not inherit a resource store and therefor depends on communicating with the Kubernetes API-Service, in order to get access to secrets and ConfigMaps. As well as the former, Fission supports a large number of programming languages such as NodeJS, Python 3, Go, JVM (Java), Ruby and many more. For further information on any of the respective frameworks, the official documentation should be considered. Also, CPU and memory resource limits can be configured, ceding great control to the developer. Two aspects which make Fission unique compared to the other three 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 pre-warmed pods can be utilized to mitigate the cold start issue, the Workflow-System makes it easier to handle an increasing number of functions. Besides that, the Workflow-System facilitates scheduling and executing more complex workflows as stated by Kritikos et al. [**kritikos2018review**]. In contrast to Fission, Kubeless uses event chaining to create workflows which makes it harder to maintain the workflow graph, when an application keeps on growing [16].

OpenFaaS is the third of the investigated frameworks. It provides a 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-memory*. Each function will be packaged into a docker image and deployed via the OpenFaaS CLI. With its API-Gateway, OpenFaaS can respond to any event, and by making use of its auto-scaling functionality the min/max amount of replicas can be configured[17].

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 IBMs Composer. As with the other three frameworks, language runtime support is quite large, ranging from Go over Java and Python all the way to Nodejs[18].

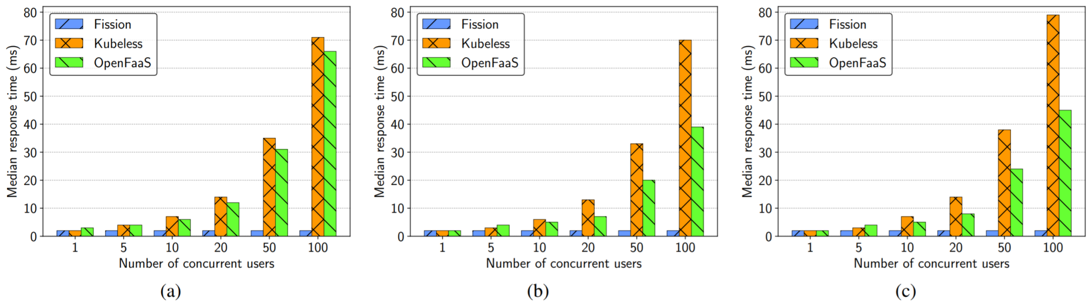
[16] https://docs.fission.io/

[17] https://www.openfaas.com/

[18] https://OpenWhisk.apache.org/

In addition, all four open-source frameworks natively support three different types of triggers, see Figure ??, namely HTTP, Event and Schedule. Adding to that, Fission, Kubeless and OpenFaaS are written in Go, OpenWhisk in Scala.

Looking at the four open-source frameworks, besides their supported languages, triggers and other characteristics, concurrency plays an essential role. Key to FaaS is its inherent and scaling nature which some frameworks cope better with than others, as shown in Figure 7. The data presented is a study from Mohanty et al. which have evaluated three of three of the before mentioned frameworks above. With an increase in concurrency, the velocity of a framework increases, which will result in less latency. The effect of startup time will get mitigated if many instances can start in parallel.
Concurrency tests on Kubless, 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 number of replicas present and the number of concurrent users [**mohanty2018evaluation**]. The test was conducted with 1, 25 and 50 replicas. Kubeless and OpenFaaS experience an exponential increase in response time. Although the response time increases and is higher compared to Fission, OpenFaaS is showing a slight 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.



(a)

(b)

(c)

Figure 7: Median response time for each serverless framework with (a) 1 replica, (b) 25 replicas, and (c) 50 replicas [**mohanty2018evaluation**]

3.3.2 Proprietary Cloud-Providers

Speaking of proprietary Function as a Service solutions, there are currently three market leaders [**kumar2019serverless**] with AWS, Azure and Google, respectively. By offering a vast amount of tooling and services, it can be challenging to choose one of these three, even though their underlying model does not differ much. By all three providers, functions are billed on the number of incoming requests, the time a it takes a function to process the request and the memory allocated to a function. The time a function runs is billed at a tenth of a second.

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, five minutes are the maximum by default, which can be upgraded to ten minutes if needed. Google Cloud Functions comes last, with a standard execution time of one minute, which can be raised up to nine minutes. Looking at the limits on concurrent instances, on AWS Lambda, the limit varies between 500 to 1000 depending on the region. Independently, the burst concurrency limit can reach a maximum of 1000 to 3000 concurrent instances. Those 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 number of functions, 1000 concurrent running functions can be reached quite fast [19].

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 compiler language, with its JVM, Java functions require more time to execute than an interpreted language like JavaScript. Tests conducted by Manner et al. 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 starts on AWS and 9.8 (JavaScript) and 24.8 (Java) seconds on Azure Functions [**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, 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 to performance ratio [**pawlik2019performance**]. Even though the performance-

[19] https://aws.amazon.com/lambda/

memory ratio is stable, the maximum CPU-performance increase seems to be reached with 2048MB of memory. 3008MB exhibits only a slight improvement on AWS. IBM and Google experience similar ratios but not as stable. In terms of delay, on AWS it was investigated that with more memory capacity the delay, when starting a new instance, decreases. On the other hand Googles start-up delays are generally constant and do not seem to be impacted by higher or lower memory sizes. On IBM tough, it seems that by requesting larger amounts of resources the time to consume the instance or resources, increases [**pawlik2019performance**]. With regard to function throughput, which is an indicator of concurrent processing capacity, 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 time.

Lastly looking at the runtime overhead of AWS, IBM, Google and Azure, on AWS it is almost evenly low regardless of a 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 Nodejs [**pawlik2019performance**].

3.3.3 Orchestration and Concurrency

Another point to consider, when migrating parts of existing architecture to Function as a Service is orchestration. The orchestration solutions vary between different cloud and open-source providers. The respective orchestration tool decides upon performance when executing functions in parallel and upon runtime overheads. Moreover, pricing needs to be considered when configuring concurrency options across different providers. As mentioned before, depending on the memory size chosen, the performance can be increased, but pricing will increase as well. Whereas in small projects, concurrency and pricing might not be the main issue, in medium to large-sized projects, memory and pricing of the various providers should be taken into account.

As stated in section *Benefits and Drawbacks* under *statelessness* when chaining functions, passing state from the sending to the receiving function is key for intercommunication. With a higher speed of state-transfer, latencies are reduced. Only with a seamless interaction of functions, complex applications can be built on a larger scale. Looking at proprietary cloud providers, the two largest platforms, AWS Lambda and Azure Functions, will be depicted and further investigated. On the side of open-source frameworks, OpenWhisk is taken into consideration.

Lopes et al. have investigated on the named frameworks in greater detail and detected significant differences in state transfer on both sequential and parallel orchestration. For the tests conducted, all functions were executed on warm instances/ containers, to mitigate inaccuracies caused by varying cold-start times, see [**manner2018cold**] and [**jackson2018investigation**]. In terms of executing sequential functions, called sequential processing, IBM's Composer and ASF proved to be significantly faster compared to ADF. The overhead, which describing the time that was not used to carry out any processing of business logic inside a function, was 1.1s for IBM Composer and 1.2s for AWS Step Functions with 40 functions connected in series. Azure Durable Functions, on the other hand, took 8s for the same 40 functions. In further implementations with [5, 10, 20, 40 and 80] it turned out, that IBM Composer only supports the orchestration of functions up to a number of 50 sequential functions. Anything beyond that limit has to be controlled by a third party orchestration tool [**lopez2018comparison**]. ADF and ASF, on the other hand, can define workflows which can run for days and months.

The evaluation of the overhead for functions connected in parallel was carried out for ASF and ADF, solely. Starting with five functions and, as described above, scaling up to 80, the results showed a significant increase in overhead of Azure Functions. With a total of 80 functions, Azure Durable Functions with an average overhead of 32.1s had almost twice the volume of AWS Step Functions with an average overhead of 18.3s. The results also suggested that ASF is more reliable than ADF in predicting the overhead to be expected. Microsoft's overhead has not always grown exponentially like Amazon's, making it difficult to predict future behaviour.

When evaluating the suitability of parallel function execution, IBM's Composer dropped out of the test portfolio right from the start, since parallel function execution was not supported as of 2018 [**lopez2018comparison**]. Meanwhile, 2020, IBM includes support for parallel execution of functions via Composer. Furthermore, IBM states that parallel execution is not restricted to a certain number of concurrent functions and can be configured as needed [20]. However, it is explicitly mentioned that the Composer theoretically has no limits, but OpenWhisk does. Exceeding OpenWhisk's limits of parallel execution

[20] https://github.com/apache/OpenWhisk-composer

will result in failures: „[...] 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 [...]“ [21]. The current limitation of concurrent functions in OpenWhisk is placed at 100 functions per namespace.

Finally, the three orchestration solutions were examined concerning transferring application state. Due to the state limitation of ASF to 32KB that can be passed to the next function, the same size was chosen for the other two solutions as well. This time only five sequential functions were tested. In 2018 the official limit of IBM Cloud Functions was at 1MB, being raised upon 5MB by the time writing this thesis [22]. ADF enables state transfer up to 60KB. It was found that IBM Composer and AWS Step Functions had an overhead of 175.7ms and 168.0ms, respectively, when executed without any additional payload. With payload, the overhead in ms for Composer was 298.4ms and 287.0ms for AWS Step Function, which represents an increase of 70%, see Figure 8. Azure Durable Functions stood out clearly in this test. With an overhead of 766.2ms without payload and 859.5ms with payload, the basic overhead is significantly higher than with the previous two but only increases by 12% under load [**lopez2018comparison**].

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%

Figure 8: Overhead bei 5 sequentiellen Funktionen mit einem Payload von 32KB, nach [**lopez2018comparison**]

As shown by Lopez et al., AWS provides the most mature solution in terms of orchestration. Not only for sequential but also parallel execution AWS provides long-lived and short-lived function chaining. Also, the limitations on state transfer to 32KB enable AWS to provide clear information on pricing, compared with the other frameworks tested. When FaaS is mainly used for lightweight function chaining tasks, OpenWhisk in conjunction with IBM Composer is a suitable solution. With its limitation on chaining functions only up to 50 in number, OpenWhisk is slightly faster than AWS in that lower

[21] https://github.com/apache/OpenWhisk-composer

[22] https://cloud.ibm.com/docs/OpenWhisk?topic=cloud-functions-limits

segment. For scheduling long-term running sequences, running for days or even month, OpenWhisk can not be recommended. If the focus is on the exchange of large application states between functions and more significant latency can be accepted, Azure is capable of such a use case. With a capacity of 60KB it its, by far, ahead of its competitors. Even sequences that run for an extended period can be configured. Moreover using async/await syntax with sequential function chaining and fan-out/fan-in[23] for parallel execution, simplifies development over solutions from AWS and IBM [**lopez2018comparison**].

3.4 Selecting 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, as described by Bardsley et al.. 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 [**bardsley2018serverless**].

Considering these additional steps of migrating a service, written in an OOP language, it might be more rational to start with a service written in a functional programming language, see also [**leitner2019mixed**]. 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, primarily being functional 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 permanent high traffic. Morover, its size and dependencies should be as small as possible as well, which will ease migration and reduce the hassel of decoupeling it into many small

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

functions.
Besides that, if an existing microservice infrasturcture is present, monitoring tool should be taken into account when depicting a suitabel service. Criteria which make a service attractive for an initial migration are services written in a functional programming language. In addition the service must not be permanently under very high traffic, because latencies, see paragraph *2.4* as well as pricing, see paragraph *3.2* will be higher compared to a PaaS solution. By incorporating the existing monitoring tools, services running most of their time in IDLE and receiving requests very frequently can easily be detected.

3.5 Effects on development

During development, teams are working according to certain process models, trying to improve their efficiency. The process models can differ slightly from each other, but two major differences are made between iterative and sequence, as marked in the following paragraph. Therfore different tools and frameworks are used to enhance productivity and improve working velocity. Some are utilized in order to maintain the platfrom, to not lose track of its present state, others apply on the actual coding/ service-development process. Nevertheless with changes in technical concepts new operational and organizational cahnges evolve. In the following, several effects of Function as a Service on the existing structures will be investigated on.

3.5.1 Impact on Teams

When a microservice architecture is prevalent, FaaS can be adopted with fewer changes than a monolithic architecture [**fox2017status**]. With a microservice architecture, 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 developing microservice with 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 great modularity compared to microservices. Persuing proceduresfromolder waterfall driven projects are still prevalent, the implications of the sequential process model would not satisfy the needs 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 still remaining to acquire experience [**battleson2016achieving**]. Serverless development will enhance the importance of continuous delivery and continuous integration. Integrating procedures like unit testing and integration testing, to diminish time to market to a minimum, need to be further investigated. 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. To prepare for integration, slow processes have to be identified and either speeded up or their effects mitigated. This way, bottlenecks will be eliminated.

Furthermore, the size of the teams, to be more accurate, the number of developers working on a function, is likely to experience a decrease. Whereas now most likely several developers are working on a microservice, this number likely gets reduced to only one developer for each function. With the separation of concerns and further decoupling functionalities of a service, FaaS naturally decreases complexity, at least on the code level. 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 [**shafiei2020serverless**].

3.5.2 Version Control System

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 function, the odds are in favour of experiencing an overhead in version control and keeping track on the application landscape. Hence teams pursuing this approach are facing duplication in their code base, because the 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, there is accompanying complexity, while an application keeps growing. Not having a clear overview of the codebase and especially a potentially vast amount of functions redundancy is likely to increase. The consequence of that is inefficiency while maintaining and expanding the application landscape. This is in a sort analogous with the scripting practice in larger software projects. If a quick fix for incompatible components is needed, a script is often used to cope with the issue. The results are myriads of scripts, that are hard to maintain because the next problem is likely to be fixed by another script. In practice, the scripts are not in one place, probably even stored in the repository of the service itself. Providing they are strongly context-bound, this is not a big issue, but many time they are of a far more general nature. Resulting in a lack of reusability, redundancies are faced and efficiency is reduced [*internal source*].

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 present VCS of the 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 separated repositories for different teams will simplify the use of many accounts for development and production, as discussed in a later section.

3.5.3 Programming Language

Another issue that can emerge during the process of development is affected by the programming language used. Looking at Java, which mainly is an object-oriented programming language, even though concepts such as Lambdas, added in version eight, the question of refactoring arises. However, besides these Lambda expressions, the core of a Java application is OOP-based. Migrating microservices that are Java-based to FaaS, it is necessary to map Java classes to functional units that can be deployed onto the serverless platform. Moreover, concepts like empty methods, getters, setters, constructors and singletons have to be taken into account [**spillner2017Java**]. Looking further, the absence of conventions outside the actual code, like the usual presence of a *src*-folder, have to be investigated. During the process of transformation, the Java-based services need to be split up into bounded contexts allowing the business logic to be separated.

With regard to the effects of many dependencies, a reduction of mutual dependencies has to be made to decreasing startup latencies. Two functions should not load the same dependencies if not necessary. This way, startup latencies will be kept to a minimum. Spillner et al. suggested an approach to automatically transform Java code to AWS-Lambda functions, consisting of six steps.

1. Conducting a static code analysis based on the service code in the VCS.
2. Decomposing the code into context-bound parts.
3. Translating the split code into functional units which are uploaded to a target repository.
4. Compiling the code and inspect issues and failures and upload the code to the binary repository.
5. Deploy the datafromthe binary repository to AWS
6. Test and verify the code.

Even though the last two steps are Lambda specific, the former ones provide a generic starting point. Unfortunately, this concept, especially in conjunction with automation, is as stated by [**spillner2017Java**] only applicable for simple Java applications. 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 that is passed to a function. This way, the same underlying object structure can be equally invoked with every new instance of a function being 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**].

3.5.4 Tooling and Multi-Cloud

Literature is currently facing a lack of tooling, in terms of the variety and maturity provided by present sloutions [**Yussupov2019˙SystematicMappingStudyFaaS**], [**leitner2019mixed**]. With a share of around 80% [**leitner2019mixed**] of practitioners using the Serverless[24] framework, this tool is the most frequently used in the market.

[24] https://serverless.com/

Besides the Serverless Framework, Terraform[25] and CloudFormation[26] are mentioned as well, but their appearance is far less. 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. Terraform and the Serverless Framework both manage resource configuration and therefore need to be connected with the providers. The connection will be established depositing authentication data for all cloud solutions to be managed on the framework. 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 vast amount of plugins, provides, i.e. an offline simulation of AWS Lambda and the AWS Gateway [**lin2018tracking**].

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. 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 free[27]. 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 vendor lock-in.

Depending on the purpose of multi-cloud solutions, there might be higher costs than using just one provider. If functionsfromone provider need to access another's providers function, its functions will be affected by cold starts. Moreover, the physical distance between the two data centres 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, 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.

[25] https://www.terraform.io/
[26] https://aws.amazon.com/cloudformation/
[27] https://serverless.com/pricing/

Regarding the deployment pipeline, in practice, many tasks are handled via scripts, as mentioned before. Introducing serverless, special tooling for deploying a function to the desired platform is necessary and needs to be included in the existing 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**].

3.6 Effects on Testing

There are several ways to test a serverless application. A distinction is made between local unit-testing, canary releases, as well as A/B testing and integration testing. While OpenWhisk was the first framework offering local unit testing, the other providers have adopted this as well. Looking at AWS, Amazon provides with their open-source Serverless Application Model (SAM)[28] the possibility to build, test and debug the application locally. On Azure, Microsoft provides the so-called *Function Core Tools* as integration for running functions locally[29]. Even though the underlying business logic can be tested locally, there is no guarantee that the code will work the same when the function is deployed into the actual ecosystem. A reason for that is the interaction with triggers, events, databases, functions and other services which, again, can not be simulated accordingly in a local environment. In contrast to that, hosting an application on-premise or via IaaS or PaaS, it often is possible to not only unit-test locally but also simulate integration testing locally. Due to being in charge of the platform, there are often local copies of databases or message queues which are quite similar or identical to those running in production. Not only are there copies but real databases that can be accest locally far more easy, due to being in charge of the „database-service“

Moreover, it is hard to provide all configurations made on the different services locally. Key figures, such as the execution time of a function, the speed of loading all dependencies and potential latencies due to cold starts will be different on the cloud platform. Also, the unawareness on which type of server the function will be started makes predictions on performance difficult [**racicot2019quality**]. Due to the complexity being abstracted to the cloud vendor, the user has no further control over these services, ececpt of what the platform allows them to do. This restriction is not affected when performing unit tests, which is very easy, based on the small code base of the functions. With all these implications, it is inevitable to test the service on the provider's platform

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

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

to get reliable results on compatibility and performance, even though tested locally.

When testing on the platform, there are several possibilities that can be chosen form. Starting with canary releases, considering the size of serverless functions, changing solely specific behaviour becomes far more feasible. After a new version is deployed, changes can be made to the API-Gateway of the serverless platform, in order to redirect a defined group of users. These cloud then test only a new version of a function or the entire application. Without being charged for code stored on the platform, only for code being run, it is possible to mirror the entire application to another account, without experiencing an increase in pricing. On that account, the new feature can be tested. Regarding the operational overhead of the mirroring process, mirroring appears to be more comparative on major changes to the application and or a function. For patches, solving defects, or minor changes, adding new features to an application, not affecting is API, the effort of mirroring might not be necessary. Using AWS, Azure or Google, canary releases can be scheduled via the CLI. On AWS in particular, canary releases can be achieved either via *aws-lambda-deploy* or AWS Step Functions. Although the opportunity of canary releases and A/B testing theoretically exist, it is not very often used by operators [**leitner2019mixed**]. One reason cloud be the operational overhead, another the implications on performance, when not having a separate development and production account. Especially load tests are likely to affect performance on an account. In case only one account is used for testing and production, latencies can have a noticeable effect on the application, affecting end consumers. Therefore, it is advisable to have at least two different types of accounts — one for production and one for testing purposes. Despite the additional overhead, load tests can be done safely without any impact on the actual application [**leitner2019mixed**].

3.7 Implications on monitoring

Debugging and monitoring are crucial parts of any application. They ensure that the application is stable. The two metrics are indispensable in determining the health of an application during testing and later on in production. After going through the different testing environments, DEV, INT, UAT right up to PROD, the data gathered can be used as a baseline of the normal behaviour of a service. Whenever running out of memory or failing to provide a new instance of a service, the monitoring solution will, in a perfect world, alert the operator before issues in production cause severe damage. Whereas the team has full control over their servers when using a PaaS or IaaS solutions, with a container environment like Kubernetes on top, it is quite different when switching to

FaaS.

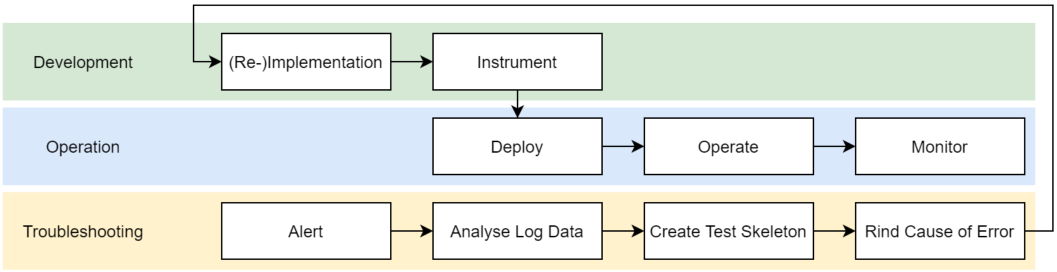


Figure 9: Monitoring cloud functions, by [**manner2019troubleshooting**]

According to several papers [**roberts2017serverless**] [**baldini2017serverless**], the field of serverless, mainly Functions 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. Not having access to the platforms infrastructure, 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, to free capacity when not needed and scale functions as efficiently as possible. 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 slightly modified way. The concept is split into three segments, namely development, operation and troubleshooting.

In the development-phase, the function should be implemented with three parameters that 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 functions are providing enough data to reproduce failures when 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 services, which collect the data logged from each function. The received logging data will be further processed to determine a corridor for the execution times of a function. Whenever a function falls underneath a predefined execution time, an event will be triggered, which informs the operations team. In DevOps there is no separation between developers and operators. The developers are, as well as the operators, in charge of the product they have built, according to the concept: „You build it, you run it!“.
With an alert being 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. Afterwards, 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 essential to log the data asynchronously before the function returns a response. After returning a response, the container will be discarded immediately stopping any processing.

Regarding open-source frameworks, it is crucial to be aware of the modularity which comes along with Function as a Service. When monitoring an environment, logging data from services and complementary parts will be collected. The data collected has to be stored within a database, being able to run aggregation, sorting and other types of evaluation metrics upon. Implementing an open-source FaaS framework into existing infrastructure, function monitoring can be included in the present monitoring solution. Being in charge of the infrastructure, metrics can be collected on a far deeper scale, i.e. the container runtime itself and as well as from the underlying OS. In case the former landscape consists of several more extensive services, potential bottlenecks need to be determined and have to be eliminated. The bottlenecks could occur in the 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 dynamically. 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.

3.8 Results

Concerning the present microservice architecture, migrating to FaaS is going to affect parts of development and operations. Due to the prevalent modularity, changes in development are minor changes, providing an agile process model, and agile development is practised. Regarding operations, significant changes can be seen, affecting the application landscape. Figure **??** depicts the observed changes of migrating to FaaS compared to the existing landscape, first presented in section *Describing the existing architecture.*

Before investigating the effects on the service landscape, Function as a service needs to be proven as a suitable solution for mitigating present flaws. Afterwards, a suitable provider has to be chosen to start with.
On the side of cloud providers, AWS is a feasible candidate to start with, when the primary purpose solely relies on performance. AWS moreover provides a vast ecosystem, which at first can be very overwhelming, considering its size. GCF, on the other hand, has fewer services that can be utilized but is easier to getting started with than AWS. If a PaaS or IaaS solution of any of those vendors is already prevalent, that platform should be used to try out FaaS. Concerning the compatibility of the platforms with itself, the hassle of workarounds to mitigate incompatibility can be avoided. Moreover, the functions can be expanded with the provider's internet service ecosystem to provide greater functionality.
If it is crucial to stay entirely in control of the system that is being used, an open-source provider needs to be chosen. The most promising seems to be Openwhisk and Fission, regarding their performance, as discussed in paragraph *3.3*. Fission exceeded OpenFaaS a Kubeless regarding concurrency as well as response time, see Figure **??**.

After choosing the right provider, a service needs to be found, which makes it as easy as possible to adapt to the new serverless concept initially. Criteria that have to be taken into account are:

1. The programming language. It should be functional instead of an object-oriented programming language. Especially when looking at languages like Java compared to a functional programming language like JavaScript, startup time is higher [**manner2018cold**]. That way, the hassle of dealing with the state and especially the objects state can be kept to a minimum [**bardsley2018serverless**]. Moreover, the concept best suits the nature inherent to FaaS, which, as the name implies, is functional.

2. The size of the service. It should be as small as possible to mitigate the time spend decoupling the different functionalities into modules, concerning their dependencies.

3. The frequency a service is called. As stated in paragraph *3.2* by Jonas et al., having a function permanently under high traffic, will make FaaS more expensive than the

same function implemented in PaaS.

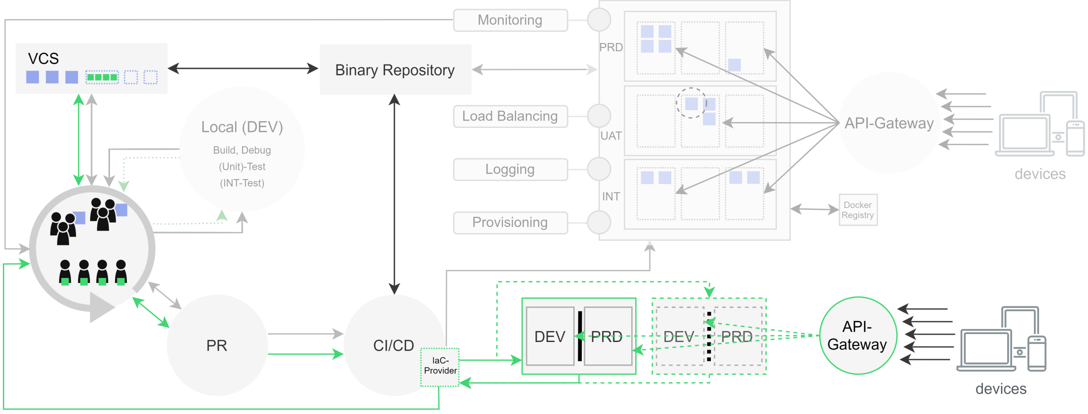


Figure 10: Modified DevOps pipeline

Figure 10, exhibits the changes on the DevOps pipeline as mentioned above. Besides the regular repositories, another repository will be added. In that repository, the different functions will be stored in, in order to reduce complexity by maintaining a separate repository for each function. For reasons of clarity and comprehensibility, the mono-repository approach is recommended over the poly-repository approach. It should enhance the quality of code and provide quick access to all functions in order to reduce redundancy amongst the service landscape.

Furthermore, the number of people working on a service, respectively a functionality, will decrease, due to the size of a function. By decoupling microservices into small functionalities, it is possible to only have one developer working on a function. As seen on the upper right, next to the development team, local testing will be affected by FaaS as well. With an increase in complexity and interaction with other services, there is no guarantee that the code will work the same in the cloud as tested locally, see paragraph *3.6*. Small code snippets that do not rely on other service, databases and are not policy sensitive can still be tested locally and are expected to work the same in the cloud environment. To guarantee code quality PRs and integration with the CI/CD pipeline will not be affected much. Significant changes were observed during the process of deployment when aiming for a multi-cloud approach, and the number of instances. Whereas with the traditional approach, three environments are used for testing purposes, with FaaS only two stages are recommended, see *Effects on Testing*. Depending on the number of services, there can be more instances to mitigate the limits on concurrent functions. That way, load tests can be performed on the development instance, without interfering with the functions running in production.

4 Evaluating the guideline

Section four deals with the application of the guideline to the existing microservice architecture. The aim is to identify possible weaknesses of the proposed actions and concerns in order to treat them adequately. Furthermore, it is to be found out in which areas the literature can already provide tangible approaches and best practices and where research is still needed. The application of the guideline is chronological, which is why the first step is to find a suitable service based on various criteria. This is then implemented with a FaaS solution and the implementation process, guided by the guide, is analysed.

4.1 Basic Considerations

This section deals with paragraph *3.2*, *Scrutinising the Decision of Migration*, *3.3*, *Choosing between Open-Source and Cloud-Vendor* and *3.4*, *Selecting a Suitable Service*, whereby fundamental questions first need to be clarified before starting with the practical implementation of the guideline. Next, the validity of the general purpose of introducing the concept of FaaS needs to be carefully discussed. To prove the soundness of introducing Function as a Service into the existing landscape, the benefits and drawbacks of open-source and cloud providers need to be considered and traded off against each other. Before starting with the actual implementation, in the last step of this section, a service needs to be depicted that will be implemented as a PoC. The PoC is not only a PoC of the guideline but also a PoC of the feasibility of FaaS, to migrate parts of the existing infrastructure to Function as a Service. Hence the different monitoring tools will be used to find a suitable service to initially implement the serverless computing concept.

Looking at paragraph *3.2* the decision on migration is scrutinised in order not to distort the possibilities of Function as a Service, by often populated benefits, as already described in section *2.4*, *Benefits and Drawbacks.* In the particular case faced at Deutsche Bank, the main purpose of Functions as a Service is not primarily to increase productivity by reducing lead-time development but decreasing costs. With the current microservice approach, services can be built relatively fast already. One to several features can be implemented in a sprint, depending on their size. Due to deployment automation by the CI/CD pipeline, a new version can be deployed in a view minutes and be set up running on an instance in the PaaS environment, in the respective testing stage. Prerequisite stays a careful code review which will be needed just as now when using FaaS. After a new version has been deployed to a particular instance, it can be tested by the developer and adjustments to the behaviour of the service can be made again. Inherent to this is the main problem Function as a Service is supposed to solve. The instance a service change was deployed to will stay up running even if the tests are over and have to be paid no matter how many tests are made. The main purpose of FaaS is to enhance the *pay-per-use* concept even further and save these costs.

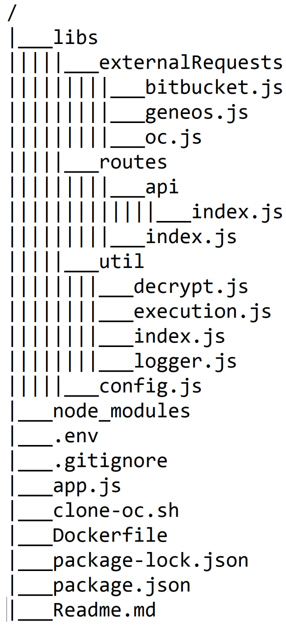
With regard to open-source and cloud providers, the latter was chosen for the purpose of migration. Looking at paragraph *3.3*, for an initial PoC, the expense of implementing an open-source solution would have exceeded the anticipated findings. To prove or disprove the feasibility, a first cloud solution was chosen over an open-source framework. The main reason was not the support of the ecosystem that the respective cloud solution comes with, but the ease and succinctness of getting started and implementing the first prototype. Moreover, contracts and utilisation were already present in the underlying architecture, which facilitated the choice with what vendor or open-source framework to get started with. To be precise, the vendor who was chosen is Azure Cloud Functions.

After elaborating the platform and being sure of FaaS being the right architectural concept, a suitable service needs to be identified. Therefore, as described in paragraph *3.4*, monitoring solutions, in this case, AppDynamics and Geneos are taken into account. Looking solely on the number of services, there are currently 39 services running in production. Twenty-eight of those services are written in an object-oriented programming language and the only one which is currently in development is written in a functional programming language. Reason to that is the different use-cases. Whereas with the majority of services, their primary task is to do calculations on transactions, process data and exchange data between different databases and systems, the service depicted aims on providing internal services. Its main purpose is to provide frontend and backend functionality, to give service owners, hosting services on our platform, the possibility to quickly access their services and perform deployment and operational tasks on them. By using JavaScript, respectively TypeScript on the frontend and backend side, a functional aspect is given. Moreover, the service only incorporates very view dependencies compared to other microservices in the service landscape. Concerning the traffic it receives, the number of requests is currently very low, due to being in the development phase. Besides, when once running in production, the estimated workload it has to handle will be very frequently, which suites the concept of FaaS even further.

The core feature of the service will be to provide an easily accessible frontend to the service owner who can deploy and promote their services initially to INT and further to UAT and PROD. Depending on their permissions, requested by authentication, the hassle of contacting one of our team members should be avoided. The team member in the following would have to look up the information of the respective lists and then promote the service to the next instance, to put in a nutshell. Furthermore, logging data will be provided and only the services accessible to the service owners will be shown not to be able to do any damage to other services.

4.2 Implementation

In this section, paragraph *3.5*, *Effects on development*, *3.6*, *Effects on Testing* and *3.7*, *Implications on monitoring* will be taken into account, in order to decouple the selected service into single functions. At first, the service will be analysed and the different functional parts identified. Afterwards, the different functionalities will be implemented each by one team member, that ideally is expert on the respective subject. This is to determine the initial threshold that needs to be overcome when introducing Function as a Service. It will also be used to determine how quickly the team will be able to cope with the new technology and what concerns are expressed about the new technology.



```
/
|___libs
|||||___externalRequests
|||||||||___bitbucket.js
|||||||||___geneos.js
|||||||||___oc.js
|||||___routes
|||||||||___api
|||||||||||||___index.js
|||||||||___index.js
|||||___util
||||||||___decrypt.js
||||||||___execution.js
||||||||___index.js
||||||||___logger.js
|||||___config.js
|___node_modules
|___.env
|___.gitignore
|___app.js
|___clone-oc.sh
|___Dockerfile
|___package-lock.json
|___package.json
|___Readme.md
```

Figure 11: Oc-execution-backend service, tree structure

Figure ?? depicts the current structure of the *NodeJS* service. The workflow is as follows. The service will be deployed to a docker container, according to the configs in the Dockerfile. When launching the container and the *NodeJS* server will be started. When the service has been initially deployed to a container and the *NodeJS* server has been started, the service will query several endpoints. The endpoints which will be queried hold information about the different versions of service in the respective environments (INT, UAT and PROD). Moreover, the endpoints provide data about the age of a service, the current amount of pods the service is running on, a region-URL to determine its position, its id and several other data that is of interest to the respective service-owner. The data of the three environments will be stored in variables on the server-side and delivered to the client when certain requirements are met. The data delivered from the three endpoints are gathered by the monitoring tool *Geneos*. At the same time the service clones a specific repository from Bitbucket, which contains central building and assembling scripts, as well as a file called *servicelist.json* which is key to the entire application. The file is maintained by the team and provides information about the current stage of the service, key- and truststore passwords, the base image name and much other information that is required to promote a service to the next instance. After the repository has been cloned into a temporary folder, the *servicelist.json* file will get extracted and the endpoints of the *oc-execution-backend* service are publically available *(internally publically available)*.

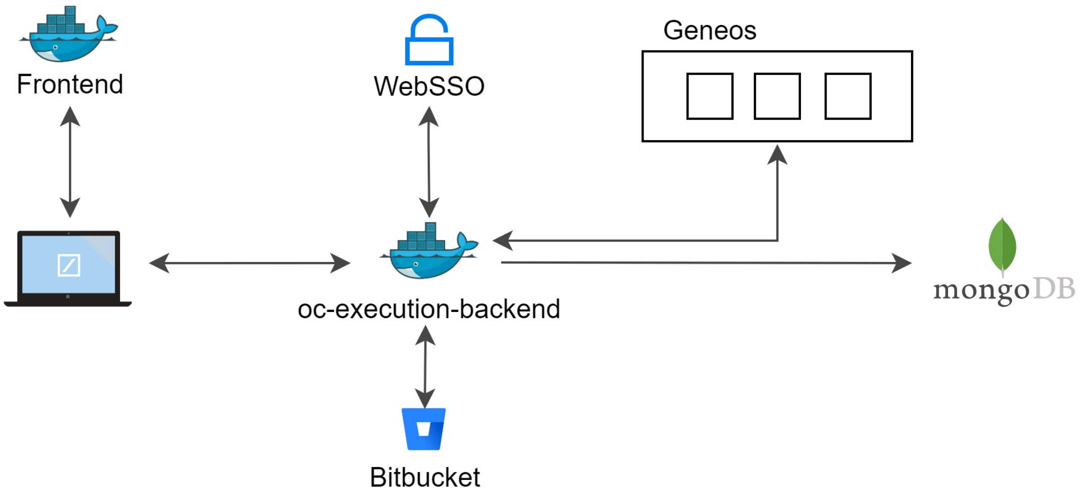


Figure 12: Implementation with NodeJS service

Whenever a client accesses the frontend, he needs to authenticate via an internal authentication service, which provided information about the user. Depending on its access rights, the services he has access to will be sent from the server to the frontend. From the frontend, built with Angular, the user can trigger promotion and deployment actions with the click of the button. If the information, entered about the targeted cluster and the token, matches the data in the *servicelist.json* the action will be fulfilled on the backend-side. To do so, a deployment script and a promotion script will be loaded from the initially cloned repository which also included the *servicelist.json*.

With regard to the guideline, the first thing that needs to be done is the decoupling of the service into smaller modules, respectively functions. Because the service, with a total of fourteen dependencies, does not incorporate many dependencies, that aspect will not be emphasized the most. Nonetheless, in order to load as little dependencies as possible, the modules can be split into different parts. By doing so, the number of modules which need to be loaded at each start of a function, are kept to a minimum; see *3.2*, *3.4* and *3.5.3*. One part incorporates modules which are required to perform actions on the database, such as *monodb* and *mongoose*. Another part includes modules which are used to interact with the underlying OS and git, in order to perform actions on the cloned folders of the repository as well as the process of cloning the repository itself. Those dependencies are *child_process*, *simple-git* and *crypto-js*, that will be used to perform operations on the tokens gathered from the *servicelist.json* and which are later on provided by the client.

Emphasizing the issue of incorporating a minimal subset of dependencies over functions, the rest of the service will be split into functional modules by decoupling its different functionalities. One functionality is the interaction with the internal authentication system, which needs to be queried each time login or actions are about to be performed. Depending on the access rights, the respective functions providing that services need to be called. The second function incorporates one of those services, which is the promotion of a microservice - one of the 38 remaining java-based microservices - to the next environment. Therefore it needs the functionality provided by the promotion script of the repository. The third functions handle deployments and the fourth interactions with the database to update corresponding records. The fifth function will provide the *servicelist.json* with promoting and deployment information to the frontend and the sixth and last function will query the general data about a service from *Geneos*, as stated above.

Due to the statelessness of the concept, a problem occurred right at the start. Whereas with the NodeJS service, the repository could be cloned and saved into a folder, that is not possible with FaaS. With FaaS, the repository can neither be stored in a folder nor can the data of a file be stored in a variable. One possibility, as mentioned in *2.4* and *3.2*, is to externalize the state to a low latency database like Redis, or a database solution provided by the cloud vendor. With Azure, the respective database would be Cosmos DB to store the data in. With that problem solved the issue of accessing the file from the repository placed in Bitbucket remained. To keep things as efficient and straightforward as possible, the entire servicelist got hosted in a function, as well as the promotion and deployment logic in the respective functions. By doing so, the hassle of cloning the entire repository to extract three files could be avoided. The three functionalities got extracted from the repository and placed each in a function. To keep track of the functions, they were saved to a functions-repository. Whenever the servicelist or the logic of the other two functions need to be changed, it can be done in the corresponding file of the functions-repository. The changes will be pushed to the repository and the updated function deployed to Azure. Hereby, the function does not need to check the entire oc-management repository, whenever changes are pushed, in order to see whether a list or script was updated.

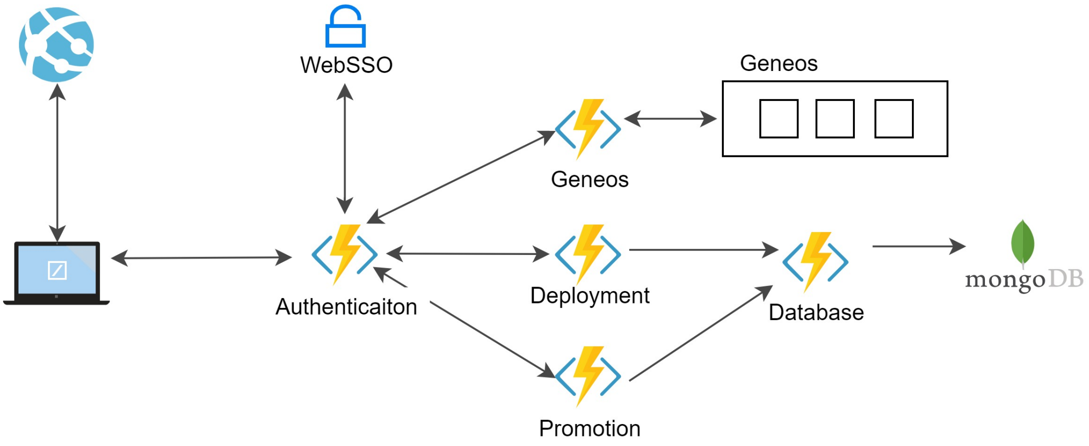
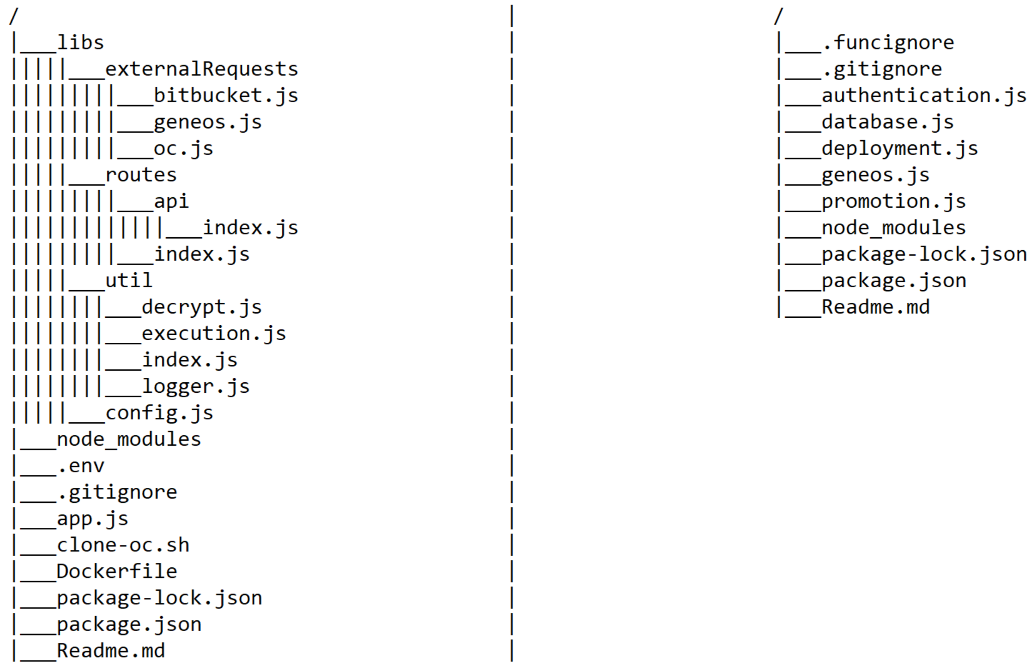


Figure 13: Implementation with Azure Functions

Figure 13 displays the structure of the service, implemented with Azure Functions. Comparing the different implementations at a fist glance, the complexity appears to be higher with the implementation on Azure. In this particular and simple example, even though the amount of components appears to be greater on Azure, the opposite is the case. As is shown in Figure 14, the amount of files is almost cut in half. However, this result should not give a false impression at this point. As also confirmed by the development team and viewed with concern, the number of functions will increase significantly when the service is expanded. Furthermore, the team noticed the flat learning curve that Function as a Service entails. The concept is straight forward in terms of solely writing functions. Nevertheless, the complexity and operational tasks of configuring IAM policies as well as integrating other services into the landscape was looked at with mixed feelings. On the one hand, the team was excited about the vast amount of service, but on the other hand, the restriction and the importance of sound security policies were viewed with scepticism.

During development, only the consideration of the stateless nature of the services required a particular acclimatization phase. The code has to be refactored, and the conceptual implementation might have to be reconsidered, in order to suit the concept of FaaS.



```
/
|___libs
|||||___externalRequests
|||||||||___bitbucket.js
|||||||||___geneos.js
|||||||||___oc.js
|||||___routes
|||||||||___api
|||||||||||||___index.js
|||||||||___index.js
|||||___util
||||||||___decrypt.js
||||||||___execution.js
||||||||___index.js
||||||||___logger.js
|||||___config.js
|___node_modules
|___.env
|___.gitignore
|___app.js
|___clone-oc.sh
|___Dockerfile
|___package-lock.json
|___package.json
|___Readme.md
```

```
/
|___.funcignore
|___.gitignore
|___authentication.js
|___database.js
|___deployment.js
|___geneos.js
|___promotion.js
|___node_modules
|___package-lock.json
|___package.json
|___Readme.md
```

Figure 14: Tree structure of Azure and NodeJS implementation

4.3 Adaptions and Further Research

Concerning the guideline, some changes need to be made in order to suit business needs better and provide greater support during the process of migration. Moreover, not only a lack of tooling as stated by Yussupov et al. and Leitner et al. but also a lack of literature on projects with a larger scale were found. Especially when looking at more complex microservices, concepts to decouple service into functions, the literature lacks tooling.

Since the service landscape consists largely of object-oriented services, a java-based service was considered for an initial PoC. After scrutinising the decision on migration, Choosing a Vendor and selecting a suitable service, starting to decouple a Java EE service resulted in an issue. Even though Spillner et al. have provided a framework for converting OOP-services into functional service, it was not possible to apply his concept to more complex services. Further research needs to be done on that field to give better advice on services which are built according to the Java EE-Standards.

According to the development team, the mono-repository approach, as stated in paragraph *3.5.2*, is in favour of the poly-repository approach. This is, to some extent, due to the complexity that comes along with a growing number of functions. To another extend the quality of code, as described in the guideline is expected to rise. Further evaluation needs to be done on the purpose of monitoring and the testing environments. By reducing four environments, DEV, INT, UAT and PROD, to two, DEV and PROD, concerns regarding the load testing were expressed. Depending on the number of functions, they need to be split into several accounts in order to guarantee sufficient capacity on each account. Due to the different limits of concurrent functions, as described in *3.3*, performing load tests on an account where several functions are running could cause (D)DoS effects. By having multiple accounts for all services, the effect on other functions can be mitigated, but the effect on pricing will endure.

5 Conclusion

This thesis provided a guideline on the purpose of migrating parts of an existing microservice architecture to Function as a Service. Therefore it has analysed various papers and articles in order to work out recommendations regarding the different steps of migration.

The guideline on migration describes a systematic approach on how to start the migration. Therefore it first describes the underlying architecture to provide a generous understanding of the existing service-landscape. Different aspects are identified and registered in an initial sketch of the infrastructure and the general process of development. Next, aspects to scrutinise the decision on migration are stated. Their purpose is to help to question the reasons for migration and sensitise on the benefits and drawback which are inherent to the concept. Subsequently, a significant aspect is a decision on a cloud provider or open-source framework, which was discussed in detail. Not only were basic considerations made, but also different frameworks on the open-source side and the corresponding counterparts on the public cloud side analysed. The most promising vendors on both sides were identified regarding the core purpose of migration and the key aspects inherent to them. Afterwards, the criteria for choosing an adequate service were given, that included size, programming language, dependencies and frequency of service calls. An issue found, which needs to be addressed in future works, was the lack of guidance on refactoring microservices written in an object-oriented programming language. Whereas the process of decoupling a service written in a functional programming language was not critical, guidance given on object-oriented services is not sufficient.

Furthermore, the guideline has discussed the issue of mono- and poly-repositories and made assumptions on a compromise between the two aspects. Effects on testing and monitoring were described in the last section of the guideline. Different approaches to testing, such as A/B-, canary release testing, local and unit-testing, were described. Besides, changes to the current environments (INT, UAT and PROD) were observed, that seemed to make the second one obsolete. Nonetheless, criticism of the reduction of the environments was made during the implementation of a service with Azure Functions. Those concerns need to be further evaluated in the following works.