2020

Adapting Microservices in the Cloud with FaaS

Mateusz Pietraszewski
Technological University Dublin

Recommended Citation

Pietraszewski, M. (2020). Adapting Microservices in the Cloud with FaaS. *A dissertation submitted in partial fulfilment of the requirements of Technological University Dublin for the degree of M.Sc. in Computer Science (Advanced Software Development)*. doi:10.21427/ftzf-yy68

Adapting Microservices in the Cloud with FaaS

Mateusz Pietraszewski

A dissertation submitted in partial fulfilment of the requirements of Technological University Dublin for the degree of

M.Sc. in Computer Science (Advanced Software Development)

2020

I certify that this dissertation which I now submit for examination for the award of MSc in Computing (Advanced Software Development), is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the test of my work.

This dissertation was prepared according to the regulations for postgraduate study of the Technological University Dublin and has not been submitted in whole or part for an award in any other Institute or University.

The work reported on in this dissertation conforms to the principles and requirements of the Institute's guidelines for ethics in research.

Signed: 

Date: **15 June 2020**

Abstract

This project involves benchmarking, microservices and Function-as-a-service (FaaS) across the dimensions of performance and cost. In order to do a comparison this paper proposes a benchmark framework.

Microservices are often known as small services which have been extracted from a bigger project and represent a very specific piece of functionality. The unit of work they represent is small enough to make it suitable for researchers to associate it with FaaS However, manual deployment and provisioning of microservices require careful steps and plenty of planning ahead.

The computation within FaaS happens on singular ‘functions’ and its main motivation is to introduce more fine grained costs, further reducing maintenance of projects deployed to the cloud and providing a more elastic architectural model, further stripping away responsibilities of more common ‘in-house’ architectures.

Thanks to the growing interest in FaaS, today's wide variety of competing open-source FaaS technologies and their commercial versions make it more accessible for developers to experiment, discover production issues quicker and work on optimizing the technology to become more mature.

With the introduction of FaaS, the process of singular functions can be optimized and potentially made better - in an architectural way, and quicker - performance wise. This in particular yields great potential when designing microservices for public and private cloud.

In this paper, the FaaS paradigm was explored and put under a test. Using common microservice usage scenarios, microservices designed as functions were put under pressure and ultimately compared performance wise to their full service counterparts. Also, costs associated with running both architectures were put into perspective for a stronger evidence of the FaaS paradigm being the next generation of microservices.

Key words: *Microservices, FaaS, Function, Service, Performance, Benchmarking*

Acknowledgements

I would like to express my sincere thanks to my supervisor Dr John Gilligan for his patience and keen mind which served as inspiration to continuously challenge my knowledge.

I want to thank all of the TU staff who worked on third party integrations, giving students access to a lot of technologies and giving plenty of possibilities and options to conduct their research.

I would also like to express my warmest of thanks to my mentor and a great friend Dr Piotr Ginalski who guided and motivated me to further pursue my educational goals.

Lastly, I would like to thank my family, and people who worked closely with me over the course of this dissertation.

Table of Contents

Table of Figures

Table of Tables

1 Introduction

This work is an exploration of two popular architectural patterns of microservices and Function-as-a-Service (FaaS) and how both technologies are applied in nowadays system architectures. Since both concepts have already existed for some time this work also aims to find how exactly microservices relate to FaaS and how interchangeable these two paradigms are when both are applied to the backend of modern web applications. Both architectural patterns are compared and utilized in the benchmarking process to be critically evaluated under the scope of performance and generated costs.

1.1 Background

Microservices are often known as small services which have been extracted from a bigger project or a solution and represent a very specific piece of functionality. As an example, in an online shop, a microservice could represent a partial search engine, so whenever a user starts typing into the search bar, a HTTP request is sent to a microservice with a query and it returns with a recommendation that appears with the option to click it. Design principles associated with microservices have their roots deeply associated with Service Oriented Architectures (SOA), often considering it as a better iteration of SOA (Pautasso et al. 2017). With the introduction of microservices, principles such as single responsibility, individual deployment and service granularity became a standard when designing microservice architectures (Gammelgaard, 2017). These principles pushed the state-of-the-art even further, leading to big companies such as Amazon or Netflix standing now as great examples of implementing microservices into existing, perhaps monolithic architectures, which are broken down for better scalability. Cost of microservices typically is based on machines they are deployed on, usually assigned with a constant hourly rate. Performance issues could include latency and complexity of operation.

This small and specialized unit of work, which microservices represent, makes it suitable for researchers to associate it with Function-as-a-Service (Kaplunovich, 2019). FaaS for short, is a technology which allows customers to execute a piece of code without the customer requiring to set up any underlying infrastructure. The FaaS paradigm is considered to exist in a bigger spectrum of serverless architectures (Alder et al., 2019).

It is yet another piece within cloud architectures and offers computation on a more ‘granular’ scale. The computation within FaaS happens with so called ‘functions’ and its main motivation is to introduce more ‘fine grained’ costs, to further reduce maintenance of projects deployed to cloud and provide a more elastic architectural model, further stripping away responsibilities of more common ‘in-house’ architectures (Yussupov et al., 2019). Cost of FaaS is usually calculated per invocation, meaning each execution of a function bears a separate cost. Performance issues could include latency, startup and complexity of operation.

Growing interest in the FaaS paradigm has caused this topic to be researched more in the past 2 years, with ‘performance’ as the most researched topic of function execution (Yussupov et al., 2019). Thanks to this interest today's wide variety of competing open-source FaaS technologies and their commercial versions make it more accessible for developers to experiment, discover issues and work on optimizing the technology to become more mature.

1.2 Research Project/Problem

The most common microservice architecture one could find in use by IT industries is the typical ‘command and query’ scenario. In such cases command usually is composed of very fast calls which aim to create, modify or delete data. A query reads any available data and transfers it back to the caller. Both of such scenarios find its use when composing larger services such as web applications (Ast and Gaedke, 2017). Therefore, if a similar system is already composed of smaller services it is very possible to utilize FaaS technology to increase performance of key areas on demand. However, moving these components to FaaS brings challenges to mentioned ‘performance’ where many may refer to as throughput, a number of completed executions in a given timeframe (Li et al., 2019). Among challenges such as prediction, scheduling and cold-start there is a problem with correctly engineering functions for cost-performance (Eyk et al., 2018). In this paper, performance is defined as throughput of processed events.

The cost-to-performance ratio is a problem as every commercial and open-source FaaS solution updates their service, be it cost or service itself, on a regular basis. Furthermore,

to give a better reason for developing microservices using FaaS, more realistic scenarios are required to give a better estimation (Li et al., 2019).

From the given problem a question arises as follows:

"Could services designed using FaaS, a serverless architecture, yield more throughput of processed events and generate less computing costs as opposed to services designed using microservices architectures?"

Throughput in this context is the number of events which were processed successfully from start to finish. The structure of the events depends on the system, e.g. for a web application this would mean issuing a HTTP or HTTPS request. This number is then put on the spectrum of time giving an overview in the form of requests per second/minute/hour.

The costs generated with the given throughput relate to machine provisioning estimated for the same time the throughput was recorded in. Cloud providers such as AWS or Azure for example, issue static costs per each hour a machine was used. Any additional technologies associated with that throughput can also be added. In the case of FaaS such cost granularity is available by default.

1.3 Research Objectives

This work aims to determine which architectural paradigm, microservices or FaaS is more suited for performance oriented backend services. A service designed with both paradigms carries out the same task that it was given in the same manner as writing out "Hello world" into a console can be done using multiple programming languages. However, both paradigms are not equal and the same way as writing into a console using different programming languages bring structural differences (in this case language itself), with microservices and FaaS there are architectural differences which may affect the performance of both paradigms.
To test the performance of both, this research aims to conduct appropriate benchmarking of both architectures. Each conducted benchmark will yield time metrics which will correspond to different actions conducted on both architectures. The time metric can then be plotted against total number of times certain actions were taken during a given

time period. The resulting tables will give a good visual view of performance based on given criteria. Including costs of maintaining both architectures into the picture, the results of this research will help individuals and Information Technology businesses in taking early steps in decision making as well as how to efficiently conduct horizontal scaling of backend infrastructures.

In this work, several objectives have to be checked in order to produce valid knowledge which will resolve which technology or paradigm might be better:

1. To explore the current state-of-the-art of microservices and FaaS architectural paradigms as well as go in depth with both paradigms to find similarities and differences.
2. To explore commercial FaaS platforms using open-source implementations such as Apache OpenWhisk and Fn Project.
3. To find out common benchmarking methods and tools to test HTTP based backend services.
4. To craft a suitable experiment using a common HTTP API backend service designed using both paradigms.
5. To configure environments for both discussed paradigms using Amazon Web Services and Microsoft Azure.
6. To benchmark designed experiments using a suitable tool and appropriately designed workloads.
7. To gather and document all time based metrics generated by the benchmark of both paradigms.
8. To analyze and critically evaluate collected time based metrics, appropriately presenting them using graphs.
9. To analyze the cost spectrum of the analyzed benchmark giving appropriate cost-to-performance ratio for both paradigms.
10. To evaluate both paradigms on collected performance metrics.
11. To give suggestions on similar problems and possible improvements for better accuracy of analyzed problems.

1.4 Research Methodologies

In order to answer the posed research question, the two architectures discussed in this paper will be evaluated under the scope of performance. With benchmarking as a chosen method of performance evaluation the two architectures will be compared against each other and undergo a critical analysis of the costs associated with running the benchmark.

Nature of the research dictates a more primary approach, as at the time this work was conducted there were no such experiments already done. The quantitative objective of this research aims to collect many metrics compiled into multiple results sets for further validation. A constructive form will ensure a new solution or approach is sought to solve the problem. Finally, the deductive reasoning of this research aims to narrow down possible answers to previously stated problems.

1.5 Scope and Limitations

Given that work of this research is strictly related to products initially released on public cloud platforms the scope of this research will remain within public clouds, focusing primarily on Amazon Web Services and Microsoft Azure commercial providers. Both microservice architectures and FaaS paradigm implementations will be covered.

Certain limitations are maintained in this study, as not every single option of public cloud providers will be covered, this relates to vast options of machine provisioning and valid arguments around the capacity of such machines.

Another limitation of this study is the budget used to have extensive testing capability of the designed experiment. The technologies provided by commercial cloud providers within the experiment are quite expensive if utilized extensively.

Additionally, literature review has uncovered the feasibility of using open source FaaS solutions. A fully set up FaaS environment by hand is difficult to achieve, especially when comparing it to commercial cloud solutions such as AWS Lambda or Azure Functions. Therefore, this research will not involve setting up open-source FaaS solutions.

1.6 Document Outline

The document is split into five distinct chapters followed by bibliography and appendices.

- Chapter 1 is a short introduction and few reasons why this research was conducted
- Chapter 2 involves a comprehensive literature review, including existing, similar approaches to solving the discussed issue using both architectural paradigms. Some gaps in the existing research are discussed and both microservices and FaaS paradigms are discussed in great detail. Similarities and differences between both paradigms are also discussed. This leads to a discussion about benchmarking of HTTP based backend services.
- Chapter 3 includes a detailed design of the experiment as well as the methodology behind it. The execution of performance tests is also discussed.
- Chapter 4 contains some of the implementation details related to the performed benchmark. It also contains an evaluation and discussion of gathered metrics and compiled results including plenty of metric graphs.
- Chapter 5 is a final discussion and conclusion of the research project, including future suggestions on the given problem as well as recommendations regarding accuracy of the experiment and its design.

2 Literature Review

2.1 Introduction

The aim of this research is to compare two architectural patterns of microservices and FaaS, by comparing performance of processed events and costs generated with that performance. To support the research, this section is dedicated to discovery of existing solutions and a detailed discussion about the two architectural paradigms in question. Firstly, the current state of research for FaaS and microservices is revised and some gaps are drawn, then both paradigms are discussed and revised in detail. A particular attention is drawn towards how traditional services are designed using both paradigms and how costs are associated with running both architectures. Then, both paradigms are critically compared to discover similarities and differences. Lastly, these differences helped in evaluating both patterns under the scope of performance and led to further evaluating both patterns under the context of benchmarking. With benchmarking as a new dimension an appropriate benchmarking framework was designed to tackle the comparison between microservices and FaaS.

2.2 Microservices

Microservices, as mentioned before, represent a very small piece of functionality taken out of a larger architecture. They are yet another architectural pattern which many consider as a natural evolution of SOA (Pautasso et al. 2017). These patterns emerged as a result and response to monolithic architectures which struggled with the greatest architectural problem - scalability. With scalability in mind, decisions to use microservice patterns often emerge as a result of business decisions (Pautasso et al. 2017).

Microservices open companies to adopt not only an architectural pattern that is good for scaling their business but also a completely different company structure model. Companies such as Amazon or Netflix utilize microservices not only to walk away from monolithic models and scale their internal architecture. They utilize them to also structure teams of developers in such a way that each team can take ownership of a group of microservices (Amazon SOA Mandate, 2011). Therefore, microservices not only

introduce architectural changes on implementation or coding level but also introduce changes to a company's business structure.

Business structure of the company is greatly affected when microservices are introduced. This very much introduces more variety of bills the company has to cover. When speaking of costs, microservices can be very expensive when deployed traditionally, especially when a project breaks down to hundreds of smaller services where some may endure heavier loads than others. In “Facing the Unplanned Migration of Serverless Applications” Yussupov et al. have researched the most common use cases of microservices which may be implemented within FaaS, also including related components to microservice architectures e.g. cache, database and messaging brokers (Yussupov et al., 2019). Initial, but outdated billing costs associated with deploying and performance testing functions have also been researched (Adzic, Chatley, 2017). This should help identify relevant places to look for when tailoring the architecture towards the cost-performance optimization and decision making.

2.3 FaaS and serverless

The term *Serverless Computing* is used to describe platforms which allow developers to run single or multi-purpose applications on demand (Hall and Ramachandran, 2019). These platforms are designed to run the application or a function dynamically and cease operation, freeing computing resources right after they have finished executing. The function in question is scaled automatically, meaning that concurrent invocations should be possible if computing resources are available.

There is an on-going discussion about the term ‘serverless’ and how FaaS is actually related to that term. Fox et al. describe serverless as the next evolutionary step from PaaS (Platform-as-a-service) and IaaS (Infrastructure-as-a-service) where all of them are part of cloud computing, an alternative to traditional bare-metal machines. However, there is no clear agreement on whether ‘serverless’ is stateful or stateless and if FaaS is limited to be event-driven only (Fox et al., 2017). Also, event-driven approaches have been the most commonly seen use cases of serverless technology in today’s research on this topic (Yussupov et al., 2019). Considering FaaS, event-driven scenarios seem to be the best fit.

Current state of research on FaaS paradigm can target very atomic areas such as security of executed functions (Brenner, Kapitza, 2019; Alder et al., 2019). Since FaaS was popularized for the cloud this kind of research aims to attract more projects which require secure transactions. Atomic aspect is assigned because such research is purely dedicated into uncompromised and uninterrupted execution of the actual function itself. However, a 'function' in this context takes the meaning of a series of functions encapsulated into one execution plan, similarly to how console applications are composed.

Many articles utilize open-source serverless frameworks such as Apache OpenWhisk (Mukhi et al., 2017) while others choose public cloud providers such as AWS (Aske and Zhao, 2018; Hafeez et al. 2018) or IBM (Nadgowda et al. 2017). Both commercial cloud and open source solutions are important to consider since underlying FaaS technology for every commercial cloud provider is a trade secret.

A recent use case of FaaS was also found in utilizing a GPU to run functions which benefit from parallel processing such as image processing (Kim et al. 2018). However, while massively parallel executions are known to be very performant, they become a particular 'taste' depending on the type of projects worked on. For example, parallelization can be quite complicated when considered in a transaction-based system, where resource locking and redundancy is more important than sheer performance.

Another recent use case of FaaS technology was found in the Internet of Things (IoT). By deploying FaaS infrastructure into smaller devices it is possible to more optimally utilize processing resources and increase lifetime in battery-based devices (Hall, Ramachandran, 2019; Karhula et al., 2019). However, these IoT solutions are somewhat against what serverless may appear to be destined for, since serverless implies that resources are constructed on demand and immediately disposed of when finished. Having many IoT devices executing code on demand may resemble machines serving computational capability, very much what commercial solutions may do.

2.4 Evaluation metrics

It was mentioned at the beginning of this paper that methodology used to evaluate both architectures will be looked at from the scope of performance testing. The two metrics which help with answering the research question are throughput and cost.

Taking a web application as an example, requesting a resource over HTTP protocol will result in the system performing some kind of action. That action is measurable not only from the system's perspective but also from the perspective of the invoker. The amount of time the invoker spent waiting for the resource, and received a successful response is called latency. A collection of such requests which happened in a chosen time frame is called throughput. This observable throughput or latency represents very accurately what the end user would expect (Vokolos & Weyuker, 1998). Therefore, to correctly evaluate which of the two architectures performs better, the throughput will be used as one of the two primary metrics.

The second metric is the cost associated with running the architecture with the given throughput. That means, when using a public cloud provider, that the cost will be associated with on-demand resources that were used. To elaborate further, for microservices the cost is mostly associated with on-demand virtual machine provisioning while for FaaS the cost can be broken down per request, e.g. AWS Lambda. (AWS, 2020). As it stands currently the pricing model for FaaS providers is aimed toward moderate throughput. It is predicted that once the throughput exceeds moderate usage, the on-demand virtual machines with microservices become more cost-effective (Eyk et al. 2018)

2.5 Existing approaches to measuring FaaS performance

The problem of performance within the FaaS paradigm has been tackled from many different angles. One very well-known problem of cold-start where functions face an increased amount of time to complete due to the execution environment not being ready has been explored and tackled by introducing a smarter package-aware scheduling (Abad et al, 2018). Another approach to this problem was also discovered to keep function executing instances alive for a certain period of time (Vu et al. 2018). Since then, similar optimizing technologies have been introduced in many FaaS providers there are today. Knowing this problem is going to be important, since migrating microservices into FaaS, will require performance of executing functions when decision making. It must transform into a strong reason for change in architecture, or migration to cloud.

Kuntsevich et al. proposed a benchmarking platform to face the performance problem using custom frameworks (Kuntsevich et al., 2018). Apart from use of performance benchmarking tools it is important to note minimizing any overhead caused by such a framework and use any provider specific technologies which place the microservice designed with FaaS close to any other technologies involved (Sewak, Singh, 2018). This means technologies such as API gateways for network routing, databases for data storage, long to short term caches, log storages and any other technologies used closely. However, these technologies also yield another widely mentioned problem of lock-in with given cloud providers. It will be important to keep this problem in mind in order to produce more reliable performance metrics (Aske, Zhao, 2018).

2.6 Gaps in research

Research on FaaS as it is today lacks emphasis regarding potential, real-world use cases of proposed solutions (Kanso and Youssef 2017). However, such widely focused research often leads to more narrow approaches and serves in a more introductory manner. Having a real-world potential use case makes this research appealing to IT businesses, especially the ones owning large infrastructures.

Currently, there is a lack of research with good comparison and educated reasoning between traditional microservice solutions which are fully built either on cloud or locally (Glikson et al. 2019). This is particularly important in supporting large businesses which no longer fit into the general problem solving community or have the capability of adopting third party solutions. Additionally, there is a growing demand to have a supportive cost-to-performance metric to conduct further research into FaaS architectures (Eyk et al. 2018).

2.7 What is a microservice?

A microservice is a service with very specific, narrow and tightly focused functionality. This functionality is often exposed by external/remote APIs to any interested parties within a system. In common practices a 'Service' can access many microservices, and a microservice should not access any other service.

Since early 2016, to this day there is an on-going discussion about what a microservice actually is (Gammelgaard, 2017). With no precise definition, the technique of microservices is adapted differently from large companies such as Netflix to enthusiastic individuals who seek clarity on the given topic. However, according to Gammelgaard (2017) it is possible to compose microservice characteristics with the following guidelines. A microservice:

- Is responsible for a single responsibility
- Is individually deployable
- Consists of one or more processes
- Owns its own data store
- Can be maintained by small team
- Is individually deployable
- Is replaceable

2.7.1 Single responsibility

In most service oriented infrastructures services are divided by their functionality or groups of most common functionalities within a system. E.g. A web-based learning portal could be composed of multiple services where some services deliver learning material to the customer while other services make sure billing of learning portal usage manages the subscription as expected. This can be referred to as a common business functionality. In case of a more technical functionality, this example may involve third party integration for which a specific service has been set up, taking responsibility of communication and is the only service which can contact this specific third party. Another way to take a view on this is to compare this infrastructure with I/O devices, e.g. a printer, scanner, monitor or a microphone where each device has its own functionality and is possibly the only one responsible for that functionality, expanding the abilities of the system as a whole.

2.7.2 Individual deployment

During development of services or any coding project, developers contribute by introducing plenty of changes. These changes need to be tested both locally and remotely, and this remote way of changing existing services or setting up a brand new

service is called deploying. In the context of deployment and how it is done, larger services may contain complex procedures and involve many steps. This becomes even more difficult if deployments have to be done after every change is done to the service. For example, in standard working procedures a project undergoes evaluation across a few environments. E.g. Development, Test, Staging and at last Production. During development it is often required to test changes somewhere where one does not have to be afraid when half of the infrastructure breaks or a chain of services yields unexpected results. This is what a development environment is usually for, to test on-going, perhaps even final changes within a specific version of a service.
But how does 'individual deployment' refer to microservices? In this case similarly to services within SOA (Service Oriented Architecture), microservices must be built as individual packages, and have the ability to be individually deployed to environment machines. This also means that within the whole system, a deployment of a microservice must happen without interrupting any other services or microservices. As microservices are deployed, the older versions are gracefully being shut down and replaced with new versions. In such cases deployments involve some form of redundancy, which means multiple instances of the same microservice managed by some kind of small load balancing system. Deployments in such cases happen one by one until all machines on configured load-balancers have the required microservice up and running - which is also important to note the fact that microservices are deployed to separate virtual machines. This form of resilience is very common now as it ensures when something goes wrong with one microservice, other microservices with the same functionality are still operating and any load coming through load-balancer is now split between the remaining microservices.

2.7.3 One or more processes

In order to ensure separation, microservices should run as separately as possible. Running microservices in the same process may be breaking the boundary and the previous principle. Another issue with having a single process handle multiple microservices is how it may resemble a monolithic architecture. Meaning that trying to deploy one microservice it may or may not break the other.
A microservice however is able to contain multiple processes. The best way to think of it is how asynchronous processing works. A microservice could, at the same time, call a

database and do some internal processing of the request as it waits for a database task to finish. Another example could involve storing a large file while some statistical results are compiled and then published to another service.

2.7.4 Own data store

The idea of a microservice arguably wouldn't be if it did not manage its own data storage. In a business scope most data require some form of persistent storage. With microservices and their corresponding data it is often handled with databases. As expensive as it may sound this form of data is relatively small. However, as more microservices are added, serving different functionalities, the amount of different data expands and becomes harder to manage. On the contrary, having multiple microservices also expands choices on data persistence technology. With this approach, it is not required to have the same kind of database for all microservices. Also, having multiple databases gives a better ability of detecting potential data persistence technology issues as data stores are upgraded or downgraded to different versions, further reducing the impact of post-deployment issues.
The ability of choosing data store technology gives greater flexibility of developed microservices. Some microservices may require a less persistent store, e.g a short term cache where data retention policy may be kept as short as a few seconds. As with all data persistence technologies choosing one solely depends on business related requirements.

2.7.5 Managed by small team

In general, microservices are not something a starting system uses. This type of architecture is used when a business/company/project grows to a point where it becomes less and less feasible to maintain by a small group of people. Therefore, when this large system is split into smaller, more maintainable microservices, a small group of people should be able to handle a good few of them.
Although, it is not entirely clear how large a microservice really is. What is known is that a team of four to five people should have enough resources to maintain a 'handful' of microservices. This team should be able to fully develop, resolve any new issues, fully monitor their microservices and be able to move them across environments in a more independent way.

2.7.6 Replaceable

Previous point discusses how a small team can manage a bunch of microservices. What is also important to note is that these microservices are meant to be fully replaceable. The team responsible should have enough knowledge and available resources to have the ability of replacing their microservice with a new one in relatively short time. Reasons for replacing microservice are various. A microservice might be written in legacy style code, and refactoring might take too much time. Perhaps the current version breaks company's or individual's standards of writing microservices. Another reason might be security or simply change of language or deployment environment. No matter the reason the team responsible should have enough knowledge of existing microservice to support the transition.

2.7.7 Costs associated with microservices

As an exact cost measure, microservices aim to generate costs based on how long they occupy a virtual machine. Given that microservices take residency in a virtual machine for much longer than FaaS would, costs are typically calculated per hour. For example, in both AWS and Azure each of virtual machine types also called 'tiers' have a separate, static cost per hour of usage with no additional costs included.

With principles, costs and possible integrations in mind, the next section will dive deeper into how microservices are structured and how they fit into various architectures.

2.8 Microservice architecture

When looking at microservices using previously described characteristics, it is possible to come across an architecture of a singular microservice similar to the one in Figure 2.1.

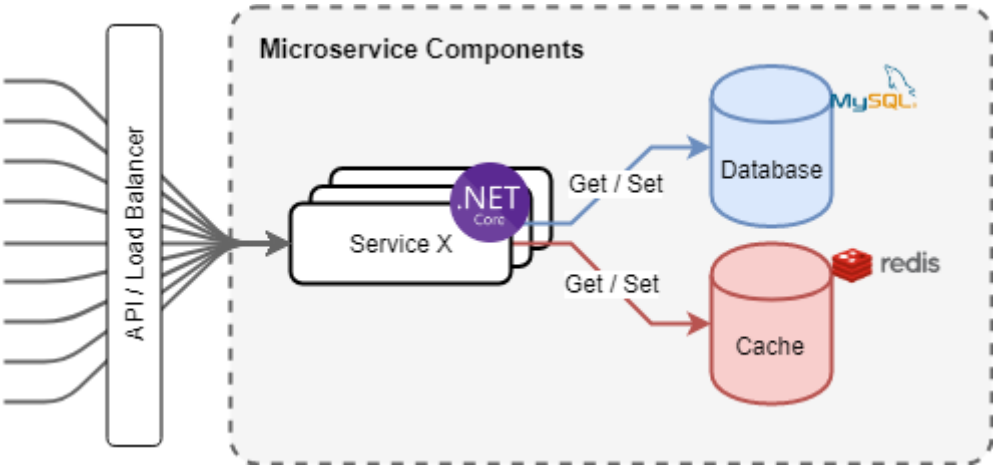

Figure 2.1 Microservice architecture

Each microservice is typically served by some kind of load balancing system. There are few reasons for this. In the future when the company grows, more instances of specific microservices might be required to handle the incoming load. This decision comes mainly from the monitoring knowledge of this specific microservice. As the load increases more instances need to be added, meaning more virtual machines need to be provisioned, monitoring needs to be updated with additional instances of the microservice and any additional issues arising from scaling horizontally need to be resolved.

Another reason for scaling horizontally might be that the company simply requires resilience of deployed architecture, and that when problems arise with specific instances where the microservice is deployed can be resolved without any significant interruptions.

In the previous section a simple use case of a microservice within a web-based learning portal was introduced. With current knowledge of microservices such architecture could be incorporated into the diagram presented in Figure 2.2.

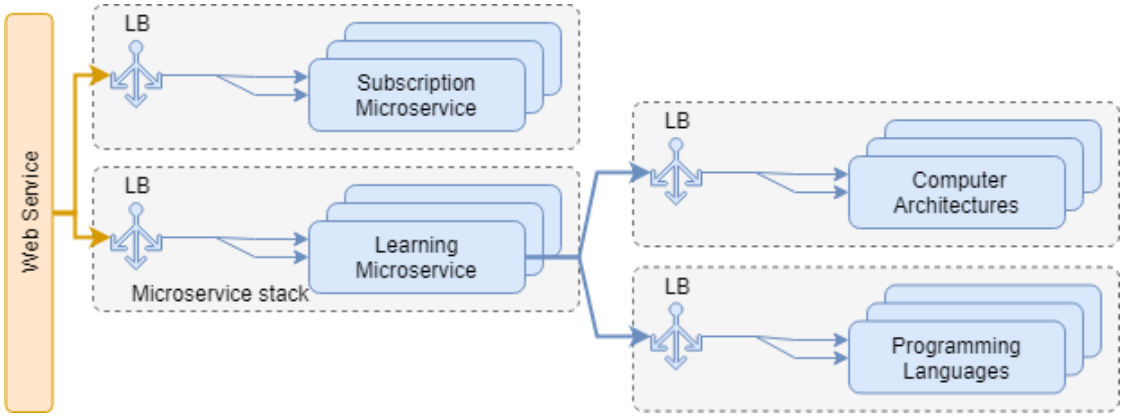

Figure 2.2 Chaining microservices in web-based architecture

In a scenario of a learning portal, it is possible to imagine a service responsible for content delivery. This service could in fact use multiple microservices chained together for the most optimal content delivery. Since most of the learning material could involve code examples, textbook samples and pre-recorded video, processing such amounts of data could have a lot of impact on the overall structure of the web-service. Hence why a microservice architecture such as this one could yield better content delivery. Of course, this only concerns services which are used heavily to the points where microservices actually make a difference.

2.9 What is FaaS

Function-as-a-Service or FaaS for short, brings the idea of provisioning ‘functions’ in the form of a service provider. A subscription-based service or an individually deployable technology stack composed of everything that is necessary to provision the ability to configure, deploy, and invoke assembled functions. In short, FaaS provides the ability to execute pieces of code, without requiring the user or a customer to know anything about underlying infrastructure, load-balancing or machine provisioning. All required knowledge falls into the chosen language runtime environment. Similarly, to microservices, FaaS is meant to be used within an event-based system, where a function is triggered by an interested party, or responds to something happening on the infrastructure. For example, an event is raised and put into a queue, the queue is a system on its own - like a broker, delivering messages to the destination specified by the caller. Each event in the queue is triggering appropriate functions within FaaS technology resulting in executing chosen processes.

Functions within FaaS are nothing more than stateless, short lived code snippets that are run on the platform. Or at least that's what is more commonly agreed on as some may view these functions as stateful as well (Fox et al., 2017). They can yield results and can also be executed without providing any results back - or the business logic could lead into results being insignificant, similar to void functions within various languages. In general, most of the functions in FaaS have some form of feedback upon which the infrastructure setup can be safely discarded after it is used.

With FaaS a common term to observe is serverless. An idea of temporary, on-demand, potentially unlimited resources, and most importantly resources which are hidden away from the operator. In general, with serverless architecture, the business mainly focuses on the costs of using the architecture which the business does not have to explicitly maintain. It spans from physical machines down to the operating system and the most a user has to worry about is the runtime environment of the code to be deployed. While FaaS and serverless have a lot in common it is important to note that resources to run serverless architectures are not absent and hence the term 'serverless' may be misleading as these resources are only abstracted away (Yussupov, Vladimir, et al. 2019). To further strengthen the point, what provisioning of machines in the cloud means is more than just deciding whether the machine runs Windows or Linux operating systems. The choice of the number of virtual CPUs, RAM and storage is still visible. This partially includes FaaS technology too, in AWS Lambda a user gets the impression of unlimited computing power where in fact every user is limited by execution concurrency (AWS documentation, 2020).

Executing the idea of FaaS reached its origins in late 2014 when Amazon Web Services introduced Amazon Lambda. It was the first commercialized FaaS architecture where customers were allowed to upload a code package or write their own code solution in a web-based code editor and execute it as soon as it was successfully created. The results of functions could be inspected with a provided web-based console. At later stages AWS provided means for integrating these Lambda calls to be executed by various events coming from other services such as Amazon S3.

It is well known that FaaS was meant to be the next step in computer architectures, further stripping away responsibilities of a system operator to manage the underlying

structure of a system. With Amazon Lambda or Microsoft Azure it is important to observe that using FaaS brings all which this infrastructure was designed for. However, it only yields the most benefit if other services provided by AWS are also used, creating a tight loop and forcing systems to be redesigned to more feasible versions.

2.10 FaaS architecture

The inner-workings of AWS Lambda or Azure Functions are not well known. However, with technologies such as Apache OpenWhisk and Fn Project it is possible to discover more about FaaS architecture and perhaps draw conclusions and speculate how commercial versions can be similar. The following Figure 2.3 depicts an overview of the programming model found in Apache OpenWhisk architecture. Interesting to see is how the main 'function' part of the model is very much the last step - in this case called actions. It makes sense, since in general, functions within such infrastructure are meant to be scheduled, run, and yield some kind of result back to the caller, if at all. Fire-and-forget types of configurations are also possible.

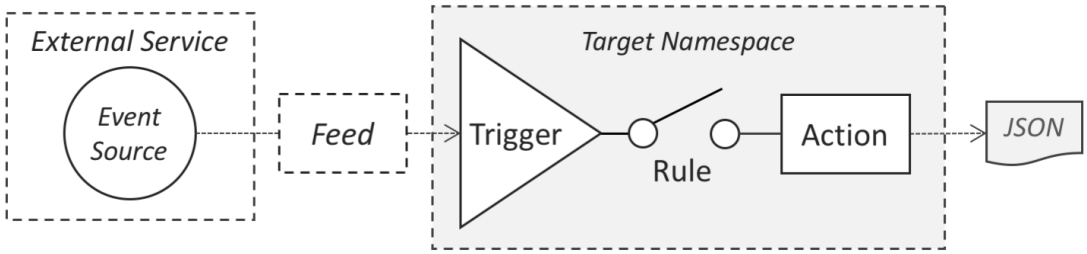

Figure 2.3 OpenWhisk Programming model (Apache OpenWhisk, documentation, 2020)

Similarly, to Amazon Lambda or Azure Functions, Apache OpenWhisk also resembles event-based architecture, where events can originate from multiple places such as previously mentioned message queues, datastores, sensor hardware, web applications and so on. In Apache OpenWhisk these events are fed into and filtered by triggers, which are configured before functions or actions in this case - can be executed. Triggers are controlled by rules which associate them with actions. When an event comes through the pipeline and an appropriate trigger is found, only then an action is executed.

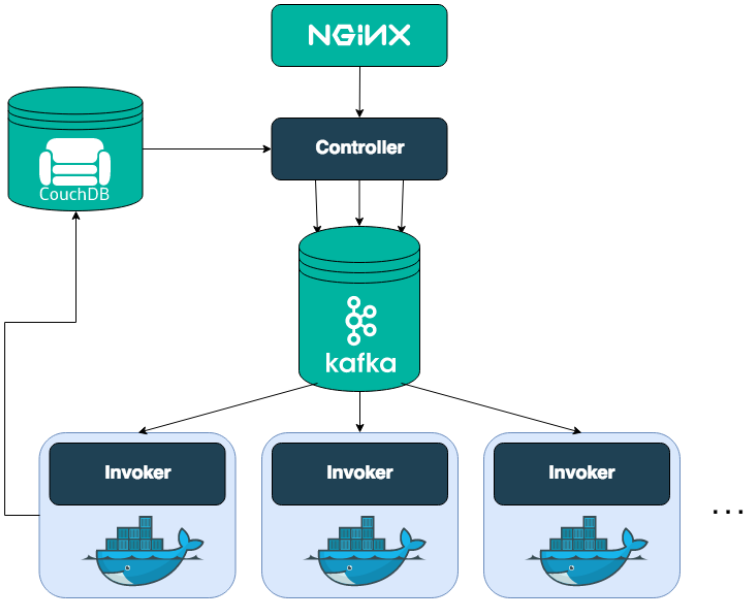

Figure 2.4 High-level OpenWhisk architecture (Apache OpenWhisk, documentation, 2020)

Another high level overview of Apache OpenWhisk is presented in Figure 2.4. The inner-workings of this architecture can be explained as follows. Any events coming through Apache OpenWhisk go through a load-balancer. In this case NGINX serves this role (NGINX, 2020). From the load-balancer events make their way to Kafka, a message broker developed by Apache. From there on any actions which are invoked are executed on Docker containers and any statistical output from function invocations, or extra logging done is inserted into CouchDB. This architecture truly shows how open-source Apache OpenWhisk is, utilizing many different technologies together forming a FaaS infrastructure.

Another FaaS infrastructure worth looking at is the one introduced by Fn Project (Fn Project, 2020). Figure 2.5 depicts how Fn Project technology handles incoming events and handles them throughout the pipeline. Similarly, to Apache OpenWhisk, the pipeline consists of multiple technologies working together. This one in particular also works with Docker containers.

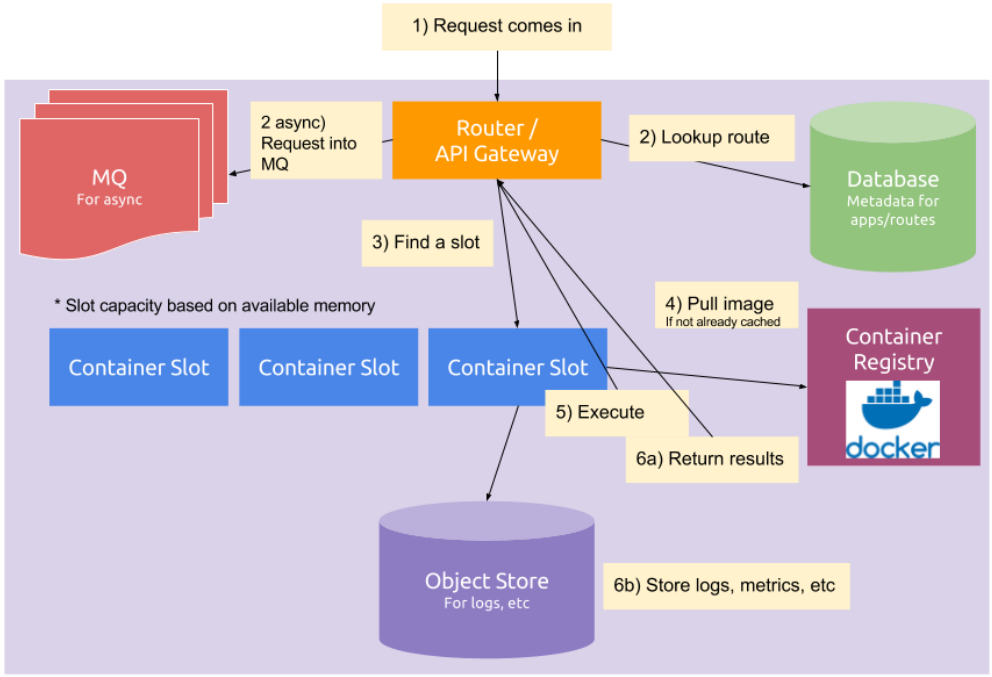

Figure 2.5 Fn Project architecture (FN Project, 2020)

In general, the Fn Project resembles a similar architecture as Apache OpenWhisk. The events are fed from a load balancer into a queue where they await their turn to be executed in docker containers. Also, any metrics or logs are stored in a separate data store. The pattern seems to repeat, the queue, containers, log or metadata store and a router to call the correct function.

2.10.1 Costs associated with FaaS

FaaS tackles costs in a very granular and easy to understand manner. Each invocation of a designed function that is run on FaaS infrastructure bears a unique and separate cost. For example, in AWS Lambda, if a new Lambda is executed, no matter the outcome, the customer pays for that specific invocation of that function. As opposed to microservices where no matter how many times a process is invoked or at all, the customers pay for each hour the machine is running.

2.11 Microservices and FaaS similarities

In previous sections it has been explored how both microservices and FaaS functions are structured and how current research has pushed the technology to be considered a good

next-step candidate in a serverless world. But what are the similarities between both architectures?

2.11.1 Event-based

In both architectures an observation can be made about an event based purpose. This could be particularly useful in the Internet of Things (IoT) where certain functionality is triggered on-demand, with sensors, small machinery or even cars. These events could also be created manually, triggered when a user clicks on an image within a web application or triggered when someone turns on a light in the living room of a smart housing setup. On a low level, the use case for both technologies is closely linked to queueing systems and message brokers, which are pre-existing or set up prior to adapting both technologies.

2.11.2 Single and specific functionality

Both microservices and functions serve a very narrow and focused piece of functionality. They are meant to be stateless and run only for a short period of time. The amount of work done in both cases vary but remains very similar. However, since microservices require a lot of initial setup, it may be possible to heavily modify the design and allow them as long running processes and potentially make them stateful.

2.11.3 Deployment

In FaaS and microservices, deployments should happen individually. In FaaS architectures such as Fn Project or Amazon Lambda this is achieved by design, uploading only the package containing code for the function. However, in microservices this process has to be entirely set up by the operating team. Whether deployment is done by hand or fully automated using third party technologies or scripts, this process is time consuming and unlike FaaS it is not present.

2.11.4 Loose coupling

As both architectures, in the picture of SOA, are destined to be somewhat independent in terms of implementation that also raises a question about coupling. Since both paradigms follow individual deployments one could assume that services implemented

using these paradigms will be loosely coupled. This is true, as it is not important where this particular service is used. If this service was tightly coupled it would mean service contains external dependencies that break the operation capability. For example, the service requires data served from another service, if such data is not present the operation fails. With microservices and FaaS, to stay truly loosely coupled, the only acceptable dependencies would be required data stores, such as database, cache or log store.

2.12 Microservices and FaaS differences

2.12.1 Runtime environment

Perhaps the biggest difference between microservices and FaaS is how the process is actually run in each architecture. In normal circumstances microservices can be run on any host machine and any operating system as long as they have all required components to function, e.g. high network bandwidth, small to medium amount of storage, good processing power and well configured firewall rules. For example, a .Net Core microservice can be run as a Windows Service on Windows host machine. The same service could also easily be refactored to run as a docker image or a Linux service. With .Net Core microservice developments became more flexible, because of self-contained code and many OS specific runtime choices. With one code solution an application can be simply pre-compiled to run on a specific machine, without the need of specific knowledge for that platform.

With FaaS it is not yet possible to reach that amount of runtime flexibility, resulting in using what the technology supports. Most FaaS technologies like Fn Project or Apache OpenWhisk prefer to use interpreted languages such as JavaScript or Python to name a few. The list of supported programming languages is limited. However, with the recent addition of .Net Core runtime to Microsoft Azure and AWS mentioned in this paper it is possible to expect more variety of deployed functions in the future.

2.12.2 Flexibility of deployment

In the previous point it was mentioned how flexibility is reduced by runtime environment. In microservices it is common to see the movement of the service from place to place. Meaning that a single microservice can be moved from a virtual machine

to another VM. One reason for doing such might be upgrading the host OS of the machine to another version with better security features. Another might be simply due to decommissioning physical server racks, another might be to simply test migration capabilities of the service. What is important is that the flexibility of a microservice allows for such actions to happen. In addition, any internal component of the microservice can be easily switched, e.g. using different databases, switching logging destinations from file to external service or simply replacing the whole microservice on the go.

In comparison to FaaS, flexibility is greatly affected by forced technology stack. In order to function, technology such as Fn Project or Apache OpenWhisk require Docker. This means that the target environment must have certain technologies in place in order to reach the potential it was designed for. Additionally, having no choice on the physical, machine layer further strengthens this point. However, this could be arguable that as a customer of commercial FaaS, one does not care about the deployment stack and the lack of visibility or ownership of physical components. This could in fact work more in favor of FaaS rather than traditional microservices.

2.12.3 Configuration and setup

With the example of a web application, setting up the service as a microservice or FaaS bears some significant differences in terms of configuration. When deploying a microservice, the configuration that is required to set up a web server bears additional requirements. One of such could be whether the service runs as a one-time process or a service that is automatically starting with every machine reboot. Another configuration could be related to opening a specific port in a firewall or installing additional software or dependency. All of these would typically be set up prior to running the service, what's important is that all of that has to be done manually.

With FaaS all of the mentioned configurations are either removed or simplified into single arguments or checkboxes within the FaaS portal given by a cloud provider. However, some configurations might be different. For example, nowadays, memory allocation is not something a developer worries about anymore, but for services such as AWS Lambda, it is important to allocate just enough memory for the function to execute without issues.

2.12.4 Load balancing

Load balancing in microservices is mostly left to the operator. It is entirely possible to set up a microservice without a load balancer. However, if multiple instances of the same microservice are spawned it is up to the developer to figure out how other services access them.
Assuming load balancing is present, with microservices, load balancing technology is left as a choice for a solution architect.

In FaaS technology such as Apache OpenWhisk, load-balancing cannot be altered. Of course, with the technology being open-source this behavior can be altered. However, with technologies such as Amazon Lambda this is not possible, and any reliability falls into trusting that the provider does the right thing.

2.12.5 Process invocation

Many microservices to this date are still deployed as services to be installed on the host machine, e.g. a windows service. This has its advantages, with one being that the knowledge of the microservice is narrowed down to function on a specific machine. This limits down runtime specific knowledge and opens up more possibilities of how the microservice is actually run or how its process is triggered. A microservice could be designed to be triggered with HTTP calls, or perhaps hook up to existing eventbus or queueing technologies such as RabbitMQ. What's important is that with microservices, all of these options are valid possibilities which do not limit developers from introducing more technologies into the architecture as a whole.

FaaS technology - in this case taking OpenWhisk as an example - supports only one way of invoking containers. One exception which might be comparable to microservices lies in the sole ability of containers. A FaaS technology could invoke completely different container images with every request coming from the load balancer. However, it still limits the ability to independently execute functions as everything is done through Docker or another container technology (if at all). Where with microservices it is possible to use a combination of both and many other machine image provisioning and data transfer technologies. In FaaS it becomes a limiting factor.

2.12.6 Service granularity

When it comes to designing microservices or FaaS there are slight differences regarding the size of the codebase. Such a decision is rooted deep within the business context which ultimately serves as a guide, driving code complexity and staying true to being loosely coupled. Compared to SOA, the number of engineers required to upkeep the codebase is meant to be relatively proportional. For example, in 2011, Amazon used to keep SOA in proportion of 1:1 to a team of 3 to 10 engineers (Amazon SOA Mandate, 2011). The granularity in Amazon left teams only with design principles to follow while creating their services. It escalated quickly with many services being developed that debugging services owned by other teams in case of issues became almost impossible and waiting for these issues to be fixed took time. This taught teams not to trust each other similarly as developers do not trust external developers.

For microservices it is suggested that the code base cannot be too small, as it may induce runtime overhead associated with creating architecture on the go. This can be an overwhelming factor which may lose the benefit of running the code on a dedicated piece of architecture (Pautasso et al. 2017). However, for FaaS architectures, the size of the codebase could be anything from one line of code to a complex procedure involving many outside dependencies (Eyk et al. 2018).

2.12.7 Differences summary

Differences between the two paradigms found in this study can bring important decisions to be made when designing an appropriate experiment. Since benchmarking brings its own complexity into the picture the differences listed in the table below (Table 2.1) also outline things that will be discussed in further chapters. This table represent a more compressed version of the differences found between the two architectural patterns.

Topic	Microservices	FaaS
Runtime environment (e.g. .Net, Python, JavaScript)	Free choice	Selected Few
Flexibility of deployment	Any compatible machine or container	Bound to internal technology stack e.g. Docker

Load Balancing	Open to any LB technology	In-built and uncontrollable
Process Invocation	Up to developer e.g. TCP, UDP, event bus & more	Chosen by internal structure e.g. http only
Configuration	Highly flexible, requires CICD set up	Less flexible, minimal CICD set up
Service Granularity	Recommended as moderate size	Can be a one-line function or bigger

Table 2.1 Microservices an FaaS differences

2.13 FaaS and microservices - purpose and selling point

If FaaS is meant to be the next evolutionary step from microservices, it is important to find out who this technology was designed for. However, to figure out who exactly is the customer of FaaS it is important to look at who are the customers of microservices.

Knowing what microservices are and how they look like in small business scenarios it is crucial to find exactly why they were made. Such knowledge will help in moving forward, as knowing the origin may help on understanding the goal. In the article written by Pautasso et al. (2017), it is deeply discussed what microservices are, what they are used for and how they relate to businesses. Some main keywords include SOA, context separation, decoupling, DevOps integration and automation.

2.13.1 Microservices as SOA but better

Most of the article written by Pautasso et al. (2017) is a discussion how Service Oriented Architectures deeply relate to Microservices. In fact, most cases describe Microservices as SOA but a more successful version, or how microservices represent a ‘best-practice’ in case of SOA. However, it is still defined as a separate architectural ‘style’, differentiating mainly with how Microservices are designed to be decentralized and independent. Also, the principles for designing microservices discussed previously also add to these differences.

2.13.2 Microservice decoupling

The article includes some deep conversations on how microservices play an important role in transforming existing architectures by separating important places into secluded

contexts. This is another point which differentiates this architecture from SOA, where components and interfaces tend to be more centralized. On a high level (Figure 2.6), decoupling in this context aims to separate clusters of similar functionalities which may be already present in a larger service. These clusters may already be partially separated into services, but because of deployment processes and internal dependencies they are still kept within the same solution or a project. Which prevents introduction of any automated deployment process as the deployment itself has to be monitored to ensure integrity.

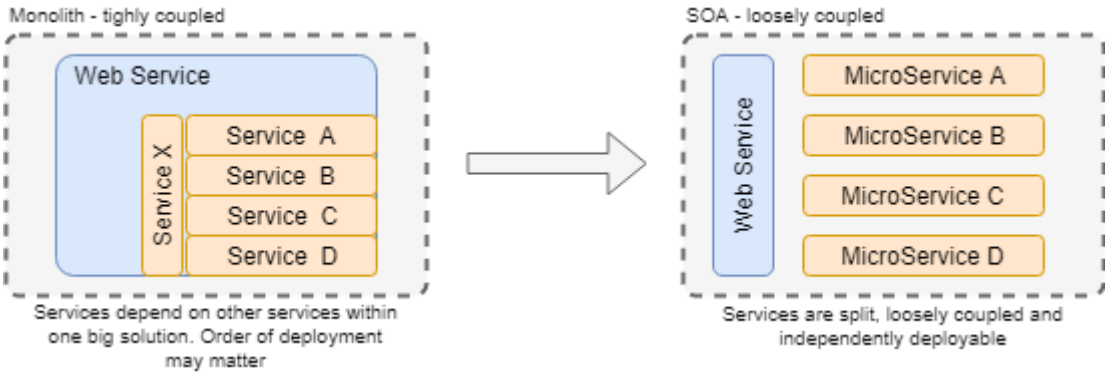

Figure 2.6 Monolithic architecture vs Service Oriented Architecture

Decoupling usually involves usage of the same communication protocols, commonly agreed strongly typed messages and similar security features to keep them consistent. The process of decoupling involves spotting a potential interface which other parts of the system could find common and using it to generate a new, separate context. This includes moving the context to separate machines, choice of programming language, security and communication protocols and defining private/public access.

2.13.3 Microservice CICD and automation

In the discussed article, it is also discussed how continuous integration (CI) and continuous delivery (CD) are making microservices more attractive, and a “fundamental prerequisite” to microservice-based architectures. This alone is a selling point to businesses who are trying to keep up with technology and the current trend of quick deployments and issuing changes faster to production.

Microservices also play a key role in automation. People from operation and development teams often called DevOps focus on how CICD creates an automated way

of deploying developer issued code changes into working environments. Since microservices are separated, the possible impact of deploying unintentional harmful changes is meant to be small and done so in a way that the team responsible can revert the change back with nothing but a click of a button. Additionally, it is said that microservices deployed this way should ease testing and validation.

2.13.4 Microservices and FaaS, selling point

Knowing the benefits of applying microservices into Information Technology type business one could say the same about FaaS. In the same way as microservices are the next evolutionary step of SOA or as previously mentioned 'SOA done right', SOA is now transforming further, allowing microservices to be defined differently in the cloud - as individually deployed functions. The main reason for doing such is actually quite simple - scalability. Because what else could be the reason for pulverizing big, working lumps of services into something that is much more granular and resilient.

Why should businesses care about these two technologies, what is the main selling point? Apart from the advantages stated in previous subsections, the article written by Pautasso et al. (2017, p. 94), suggests that microservices are used to achieve business goals, and if these goals cannot be tied with IT practices then microservices are not a good idea. This also relates to an important note that microservices is not an architecture a business starts with to prove a business case. A business needs to be already well established and has to have dedicated development teams to support future innovations. Microservices can be applied only then, if the business case is valid and the supported infrastructure is no longer able to handle customer load and of course if there is a team to support development. The best example of this is to imagine a starting web-based shopping system such as the one created by Amazon. As the popularity grows, so does the need for more scalable architecture and that is where microservices might be considered.

If microservices require so much infrastructure to be set up, a proper and dedicated development team and appropriate, valid business cases to be realized, then how about FaaS technology? Well, it looks like the way FaaS is advertised by AWS, Azure, Apache and others, it is meant to look easy, and require no big team to set up and manage even a large amount of functions. In fact, the technology provides means and has enough documentation to enable everything to be developed just by one person.

It is important to note that functions are not fully fleshed out microservices as described previously, because of how setting up both technologies are quite different. However, both of them provide the same benefits while FaaS also strips more operation responsibilities allowing a single DevOp to orchestrate the whole architecture.

2.14 Evaluating performance

2.14.1 Benchmarking

Benchmarking is a method of conducting necessary actions and activities in order to evaluate how a given piece of software or a system is going to perform once it is deployed and used in the field (Vokolos & Weyuker, 1998). It is also a way of assessing the system's availability. Meaning that whenever the system undergoes high levels of stress, be it increasing the number of processed requests (throughput), or high resource consumption (machine resources), the system still processes these requests or events in acceptable numbers (Vokolos & Weyuker, 1998).

Typically, benchmarking involves predicting how a given system is going to behave once actually used in the field. These predictions involve tailoring the usage of the system in a composable manner that can be reused again many times. Whether it is throughput or consumed resources, these tailored tests aim to evaluate the different aspects of the system and are referred to as workloads (Cooper et al. 2010). The word 'tailored' is used here as the workloads are designed very specifically in the context of the tested system. For example, the YCSB package designed by Cooper et al. aims to benchmark the capabilities of different database systems that were popular at the time (Cooper et al. 2010). The workloads are tailored with appropriate amounts of different database operations such as insert, read etc., which are run for a specific amount of time. The results of such a benchmark could give insights on potential limitations associated with the tailored usage of the system and lead to further suggestions in improving it.

As another example, when looking at performance testing of a web-server, a workload could represent a percentage or amount of actions performed while a specific endpoint is bombarded with requests. The complexity of servicing these requests under stress will outline if the system is capable of processing them in appropriate time and render the capabilities of the web-server.

2.14.2 Performance testing FaaS platforms

The most common observable trend among research involving FaaS platforms is performance testing (Yussupov et al. 2019). To be specific, such testing aims to optimize the performance of a certain area within FaaS infrastructure, be it function scheduling, resource management, or raw processing power. The article by Yussupov et al. (2019) explains how different types of performance tests were involved for each of these areas based on this common trend. Since research involved in this paper is not focusing specifically on inner workings of FaaS infrastructure it falls into a category where it is suggested is to perform a set of integration tests involving some of the metrics contained in Table 2.2.

Metric	Description
Throughput	Number of operations the system is handling in a given timeframe.
Latency	Amount of time taken to perform an action.
Payload size	The size of request in bytes which the system is going to handle
Workers or Threads	An allocated space where the work is done. In the context of a machine a thread is typically used to indicate the working thread which is handling the request or an operation.
Memory usage	Memory allocated to do a specific task
CPU usage	Amount of raw power dedicated for completing a specific task

Table 2.2 Performance metrics

In general performance tests are expected to be conducted even before deployment of developed services as microservices. Such tests often take the form of simple unit tests which are expanded and conducted before integrating them with other systems (Heinrich et al. 2017). This gives a better idea of what the system is going to be dealing with and how to appropriately prepare the physical environment to handle new pieces of software.

2.14.3 Benchmarking of FaaS and Microservices

A traditional way to conduct performance testing is to develop a benchmark. A benchmark is meant to represent a prediction of how a given system or component is going to be used when deployed and used on a daily trend (Vokolos & Weyuker, 1998). This approach is normally used when a system is defined, and components are linked

forming an execution plan. However, in case of research in this paper only a specific component is picked out of the whole system in order to stress test its capability to perform actions. Benchmarking in this case is suggested to take the form of load testing (Vokolos & Weyuker, 1998). Performing such a benchmark could be done in multiple ways. One way is to prepare and fabricate the load required to test prior to actual testing. That way every time the component is tested it should indicate more or less the same outcome each time the test is run. Another way is to fabricate the load as the test progresses, taking a more procedural approach. This involves the use of another component, typically a software application which generates the load and retrieves results right after.

Another problem with the architectures discussed in this paper is that the prior workload for benchmarking cannot be known since no particular application of both architectures is taken as an example. Therefore an experiment which can be comparable to known workloads must be used.

Benchmarking can be designed to target multiple stages of the test, typically monitored in application performance management (APM) tools (Heinrich et al. 2017). This can range from monitoring hardware specific, low-level measures such as CPU and memory but can also be designed to track low-to-mid level system calls, or application specific method invocations. In this research, the usefulness of full range APM diagnostics falls slightly out of scope, as the certain metrics such as CPU usage and low level system calls cannot be directly traced inside FaaS providers. The limitation of FaaS designed metrics causes the need to search for an appropriate way to compare the two architectures. Based on that problem, the comparable metrics for both architectures fall within any time or latency based executions. This means any crucial operations for both architectures should be measured and recorded appropriately in order to compare them. Such metrics could mean anything from measuring the total time of executed method or function to measuring outside dependencies such as connecting to the database or calling a third party API.

The nature of the two architectures in question allows a more granular and independent approach to performance testing. As opposed to big blobs of services in monolithic architectures which need extensive and careful testing, microservices and FaaS are more

isolated and are more suitable for stress testing (Heinrich et al. 2017). In a stress test, microservices and FaaS can be put under pressure for a short or long period of time. During and after the test the behaviour can be observed by monitoring logs or any other form of APM in order to assess the capability of the architecture.

2.15 Revision and conclusion of findings

In this chapter both microservices and FaaS technologies were deeply discussed and revised with supporting research in order to help conclude how interchangeable these two terms are.

Both paradigms are carefully designed and tailored on a case by case basis. Both are also designed to tackle the greater need for scalability in large architectures, a web-based shopping taken as an example.

Ultimately, it cannot be decided if one technology is better than the other, it depends. It depends on the following factors:

- Choice of operating system, which could be a primary choice when holding OS licenses. Possibly more suited for microservices due to hardware ownership.
- Deployment flexibility, where an existing or non-existing CICD stack introduces more complexity.
- Runtime environment, where the choice of language can be crucial if company standards enforce usage of certain programming languages or an exception is made for raw processing power or complexity.
- Cost of architecture upkeep, which is a factor for setting up every type of architecture.
- Performance of the architecture, which can vary between each drastically and must be decided through a good proof of concept and testing.
- Monitoring of the architecture, outlining which resources need to be monitored and which function or method calls need to be tracked. This also involves how much time is taken in order to perform these actions.

It has been established that in order to compare the two architectures it is crucial to design a set of appropriately designed performance tests and perform benchmarking of

microservices and FaaS designed services. The benchmarking needs to resemble a trend in usage of the service designed for both architectures. A trend can be something decided manually and adjusting the ratios of the trend should result in change of performance.

Performance testing suited best for both architectures is stress testing, therefore an appropriate benchmarking tool and methodology must be used to test the service developed as a microservice and FaaS.

These factors will serve as guidelines and help in identifying a suitable experiment in order to prove FaaS a worthy candidate and a counterpart of microservices in the cloud.

The next chapter explores how the given comparison of microservices and FaaS technologies could be put into a perspective of a testable web-server. Based on a sample idea the design will be refined and appropriate architecture will be designed using the chosen cloud providers.

3 Design and methodology

3.1 Introduction

The aim of this paper is to find out if a service designed using FaaS architecture yields better performance and generates less costs than the one designed using microservices. It was already established that in order to do that, these architectures need to be compared against under the context of performance. To start this process this section aims to design a benchmark to compare FaaS and microservices. This will involve designing an appropriate service and its surrounding architecture in order to proceed with evaluation. Solutions for both patterns are carefully designed and tested, involving plenty of automation scripts to make the process quicker. Some important comparisons are made to clearly distinguish between solutions and important reasons are given for the choices made regarding the architecture.

Architectures designed in this section were hosted using AWS and Azure cloud providers. The reason for choosing them is because they are highly available and involve plenty of free tier plans that are cost efficient and appropriate for required computational power.

Comparing a service designed as either microservice or FaaS requires an appropriate design of an experiment for the following environments:

- Microservices (AWS and Azure virtual machines)
- FaaS (AWS Lambda and Azure Functions)

To do so, this chapter outlines tasks as follows:

- Introducing the idea of a suitable service to be potentially used as an experiment and how well does it fit using both paradigms
- Designing an appropriate way of conducting the benchmarks
- Deciding on which HTTP benchmarking tool is best suited for the job
- Providing high level designs for experiments in question
- Introducing internal, remote components of database and caching into design
- Designing environments with appropriate networking and automation

- Producing benchmarking workloads including some insights regarding executions of experiments using pseudocode

3.2 Idea and example

In most web-based applications, or in fact, any public, customer facing application contains some form of user management behind the scenes. It could range from simple account creation, similarly to various social media sites. It could also be session-based, one time user set up, much like public web-based chat rooms. Also, it could be as simple as tracking potential user activity on a website using cookies. No matter the form, this common functionality involves creating an arbitrary link of the user to a virtual representation on the application.
A simple case of creating a user on a site could be managed by a microservice which manages its own user or account store (Figure 3.1). Potential use cases of such service could involve:

- Creating an account - most of the required data is set up by the business
- Modifying account related values - could be particularly useful for other services within infrastructure, e.g. adding claims, permissions
- Deleting account - in most cases this resolved by marking the account as deleted and not actually deleting records, as disk fragmentation could be another issue
- Querying for existing accounts - possibly the most used functionality
- Any other query discovering relations of accounts

Each of these actions should also have a corresponding mechanism for updating a small cache. This is particularly important in high-performance systems where data needs to be accessed very quickly. By keeping a short, time-limited cache a microservice could ensure that most frequent data is accessible, without impacting performance of the ultimate source of truth - a database in this case.

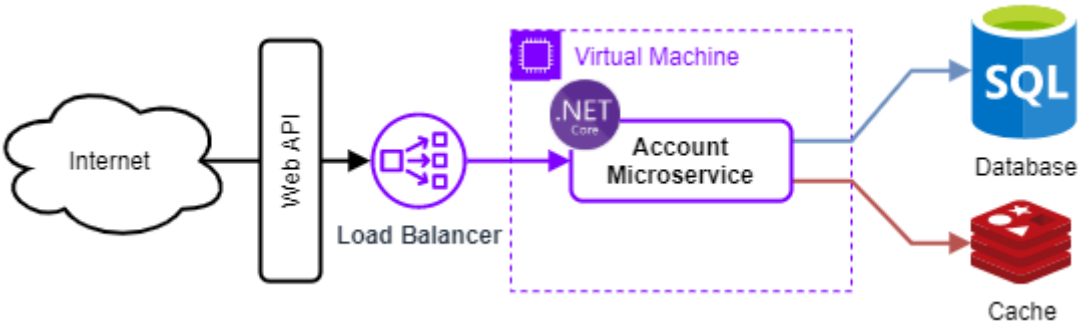

Figure 3.1 Simple account microservice architecture

Why is this a good example? Because, this microservice is only responsible for creating, modifying, and deleting accounts data and any additional information related to them. The scope of the service is only related to account data and nothing else. If this service was to throw faults and shut down for any reason, it only affects some functionality of the web application, but the web application itself is untouched. For this reason, this microservice is fully, independently deployable. This service is also fully replaceable, the choice of technology stack and provisioning process is up to the business and team who manages it. Its size should also allow for very small amounts of work involved in supporting or bug fixing.

Supposing this web application could involve a login mechanism. Without going too much into detail of standard authentication protocols and possible technology usage, previous account microservice could be used in a chain with a new login microservice. Login microservice could access Account microservice for any required information, such as the existence of the account user, and any business information which could be utilized in creating claims (Figure 3.2).

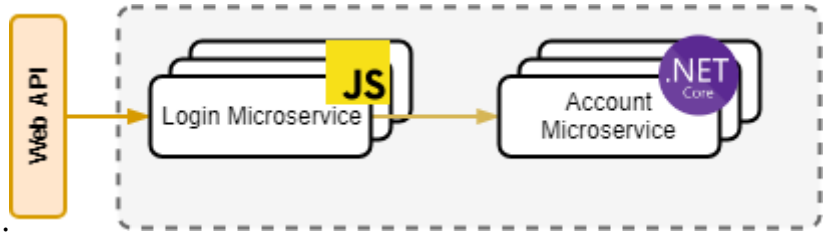

Figure 3.2 Microservice chaining

A login service could then only provide functionality around handling appropriate user-issued tokens, and perhaps application-issued tokens too.

Like mentioned before, the concept of microservices is not new, and reminds a lot of Service Oriented Architectures (SOA). However, the best practices discussed earlier make it significant enough to be treated as a separate type of architecture.

So far only a small microservice example has been discussed. It is also worth discussing how this microservice finds its place in a bigger picture (Figure 3.3).

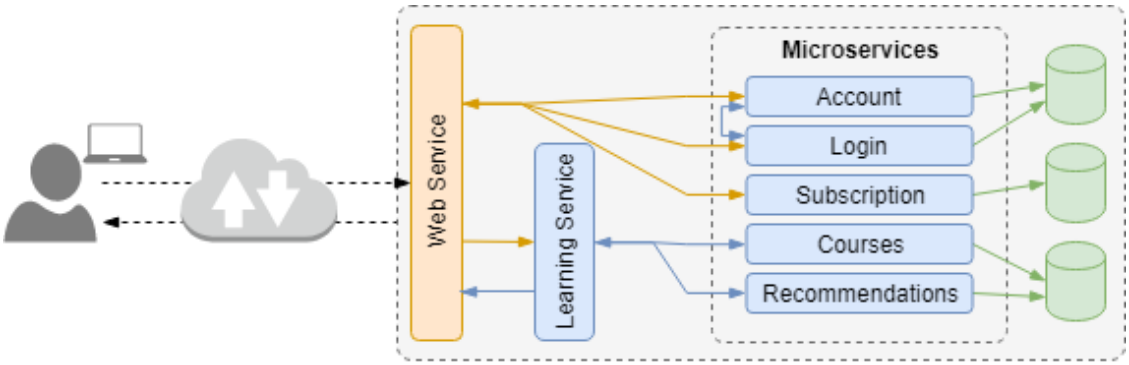

Figure 3.3 Sample learning portal architecture

From a further perspective, in a web-based learning portal, the end user only sees the content of a web page it was presented with. Behind the scenes multiple services are working together to deliver the content in parallel. Multiple databases are often put into picture too.

In a slightly bigger example, microservices are numerous and scaled accordingly to demand. This ‘demand’ is often based on monitoring microservices through extensive logging, custom metrics and load balancing metrics. Scaling happens in a horizontal approach, adding exactly the same instances of the service, deployed on a different virtual machine. The approach of scaling varies with different technology stacks, it could be either automated or manual. In a manual scenario, judgement is decided by the human eye - by watching collected metrics and is generally connected with provisioning new machines on a long term basis. The good thing about such an approach is that hosting machines are picked very carefully by the team responsible, but also introduces more maintenance. In an automated scenario this is no longer a concern, however it requires knowledge on deployment stack and any technology involved in addition.

Going back to the example with a microservice for managing accounts. It might be worth taking a look at the physical picture of how such structure might be set up in the cloud.

The example is somewhat universal, and the technologies outlined are supported by major cloud providers such as Amazon Web Services and Microsoft Azure.

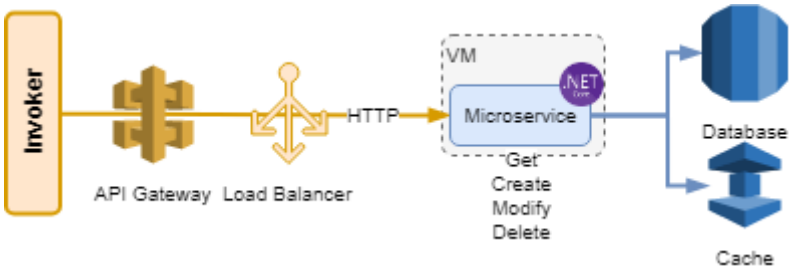

Figure 3.4 Account microservice designed in cloud

In ideal circumstances the microservice will be deployed independently on a dedicated machine. It is important to treat this microservice specially, giving it its own VM as putting more microservices on the same VM might impair performance of every process on that VM, including microservices. In addition, when trying to compare two architectures it is worth reducing further differences by designing both to be as similar as possible.

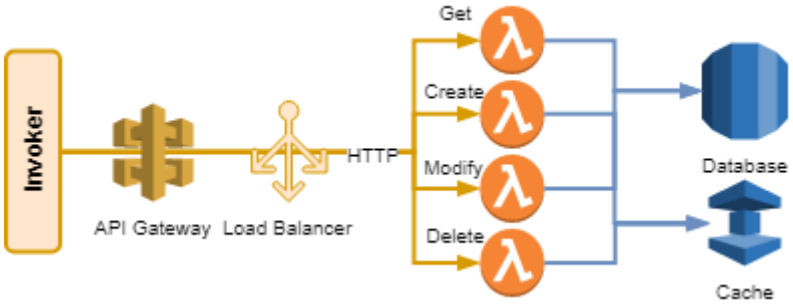

Figure 3.5 Account microservice in FaaS - first approach

When converting this architecture design to use FaaS technology instead it is possible to do so in multiple ways. First approach (Figure 3.5) could involve creating functions one to one. Meaning that each functionality of the account microservice is directly translated into a corresponding function. Designing functions this way grants services to be constructed in a more ‘fine grained’ fashion. However, this granularity also results in designing functions to be redundant, operating in a very similar way, yielding different results.

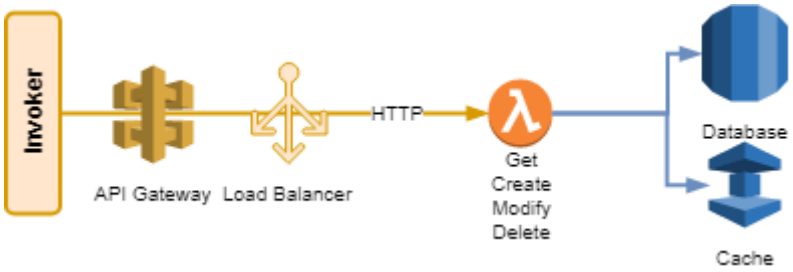

Figure 3.6 Account microservice in FaaS - second approach

Second approach involves recreating the microservice as a single function (Figure 3.6). The way it would be controlled remains very similar to the first approach. In fact, other than the binary size of the function, it is expected to have almost exactly the same performance as in the first approach. An educated guess is a difference of a few milliseconds

3.3 Preliminary design

To implement an appropriate service which can be used by both microservices and FaaS, the previous example from section 3.1 will be refined and revised to provide a good basis for conducting a performance test.

Logically, the architecture will be set up entirely in the cloud. With a personal computer acting as a trigger for the experiment (Figure 3.7). That personal machine can also grab any results generated by the benchmarking tool. The results will be generated as text files. Also, any logs generated by microservices, or functions will be retrieved either from generated log files or a common log store (i.e. CloudWatch in AWS).

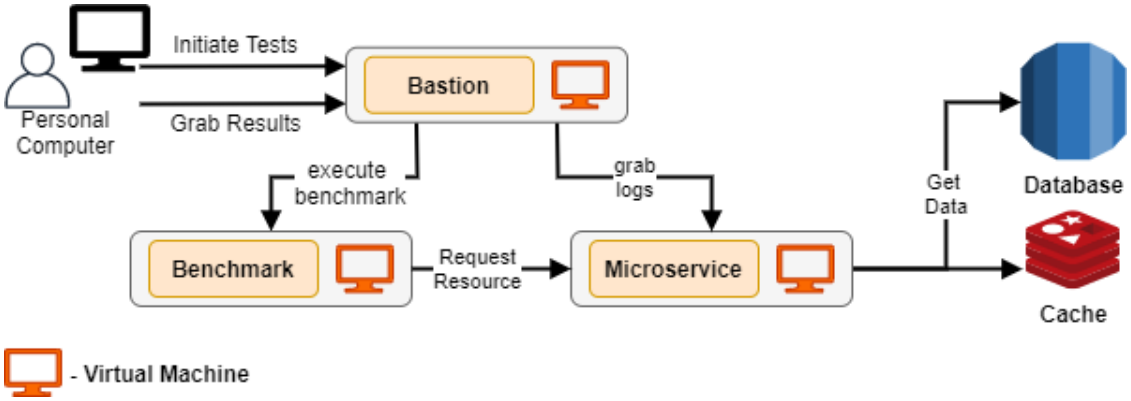

Figure 3.7 Microservice architecture logical view

A personal machine must access some machines throughout the system in a secure manner. This can be achieved in multiple ways depending on the cloud provider. For example, in Microsoft Azure most actions can be executed through built-in console and script executors directly in the browser. However, this is only possible in the browser and gaining the same ability outside of the portal requires a more custom way of accessing machines, e.g. a windows remote desktop connection.

For better security, the idea is to limit how many machines must be exposed to the public. To achieve that only one machine in the cloud can be exposed to the public and further connections can happen from that machine. It is often referred to as 'jumping' and the virtual machine used in this process is called a jump box. It is further discussed in section 3.7.

To finalize, both solutions will use the same runtime environment if available (.Net Core 2.1, version 2.1.805). The application itself will be as discussed before - an account service with multiple endpoints:

- Creating an account (POST) - creating an account into database
- Modifying account related values (PUT) - modifying data in database, clearing cache data for that specific record
- Deleting account (DELETE) - deleting database records, clearing cache values
- Querying for existing accounts (GET) - retrieving data from database if not present in the cache, saving data to cache so that next GET request can retrieve data from cache instead

3.4 Benchmarking Tools

For this experiment, creating a custom, self-made benchmarking tool which will issue HTTP requests is out of focus. Creating such a tool is a research topic on its own, ensuring correct multi-thread support and appropriate parallelization. Therefore, this experiment will utilize open-source frameworks available today. Currently, there is a wide variety of benchmarking tools available to help with performance testing HTTP/HTTPS based services. Out of all open-source benchmarking tools available today, the choice was narrowed down to only a few.

- WRK (WRK, 2020) - open-source HTTP benchmarking tool. A very raw C based tool that uses Command Line Interface (CLI) to benchmark HTTP endpoints.
- LOCUST (Locust, 2020) - open-source web-based benchmarking tool. A fully fleshed out benchmarking tool with web interface, implemented in Python language.

Type	Name	# Requests	# Fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS
GET	/	5416	0	21	21	4	38	20336	44.1
GET	/blog	1745	0	27	26	3	49	20370	13.7
GET	/blog/[post-slug]	1824	0	15	15	2	27	19943	15.9
POST	/groups/create	185	0	57	55	5	108	3273	1.9
GET	/signin	10266	0	26	26	3	49	19949	66.6
POST	/signin	10266	0	82	82	45	120	20030	66.6
GET	/users/[username]	1802	0	31	31	6	55	20194	15

Figure 3.8 Locust Benchmarking Tool Web Interface (Locust, 2020)

According to documentations supplied with both benchmarking tools, both have the ability to be run via command line utilities. This also means that both benchmarking tools are very suitable to run from UNIX based machines as graphical outputs are unnecessary for performance oriented executions. The ideal choice here for an operating system which will run a benchmarking tool will be Ubuntu Linux either 18.04 LTS or 16.04 LTS.

Both benchmarking tools also support scripting capability, allowing the user to explicitly control the tool's execution pipeline. For the purpose of this experiment the tool needs to be controlled in terms of request path and body parameters where each single request can be modified and executed without any performance loss of the tool.

Evaluating performance of the two architectures using one of the discussed benchmarking tools requires appropriate workloads. These workloads are described further in this chapter (Section 3.9)

Both tools are designed to work only with HTTP or HTTPS protocol. Therefore, they will serve a solution to issuing huge amounts of the given protocol to stress a web

application designed using both architectures. In the next section WRK benchmarking tool is discussed in more detail and a reason for choosing this tool is stated.

3.5 Benchmarking with WRK

WRK (WRK, 2020), as described before, is a modern tool capable of issuing an incredible amount of HTTP requests despite being run on single multi-core machines. With multithreading support and controllable processing pipeline this tool is an ideal candidate to conduct any form of stress testing with HTTP or HTTPS based web servers.

Installation of the tool requires following packages to be installed prior to running the tool:

- LuaJIT - WRK has scripting functionality, which is done in lua, and as such requires this package to operate properly. Also, the author explains that the tool contains code dependent on LuaJIT project
- OpenSSL - another required package
- make - a building utility for programs and applications.
- GCC - a GNU compiler system, required as the tool is built mainly in C

A sample execution from the tool can be seen as follows:

```
wrk -t12 -c400 -d30s -s script.lua http://127.0.0.1:8080/index.html
```

- `wrk` clause initializes the tool with further parameters
- `-t` parameter specifies how many threads the tool should create and maintain throughout the testing phase. Another variant of this parameter is `--threads`.
- `-c` parameter specifies the amount of total connections to keep between threads, where each thread will keep the relevant number of connections e.g. N = connections / threads. Another variant is `--connections`.
- `-d` parameter is the total duration of the test, in this case specified with 30 seconds. Another variant is `--duration`.
- `-s` is an optional parameter which specifies a Lua script the test should use. Another variant is `--script`.
- Last parameter is the URL of the site the test is going to target.

Amongst other available parameters these are the main ones the experiment will focus

on.

Project's README.md illustrates how the test is executed with above description and potential metrics provided at the end of the test (WRK, 2020):

```
Running 30s test @ http://127.0.0.1:8080/index.html
  12 threads and 400 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   635.91us    0.89ms  12.92ms   93.69%
    Req/Sec    56.20k     8.07k   62.00k    86.54%
  22464657 requests in 30.00s, 17.76GB read
Requests/sec: 748868.53
Transfer/sec:    606.33MB
```

The main metric of the test is the number of requests per second the test was able to issue during specified time. It also provides the user with some information about the latency of requests. Amongst presented values the last two of 'Max' and '+/- Stdev' are a little misleading or misunderstood. By digging into the past issues on the project's GitHub page the only information on it comes from the author which describes them as *'Max is the maximum value seen, +/- Stdev is the percentage of requests within one standard deviation of the mean'*. Furthermore, another issue thread discusses the potential use of these two values to represent a 'heavy tail' if the values of these metrics are found to be high.

3.5.1 WRK Scripting and Latency measures

WRK gives some insight on request latency measures throughout the test. Metrics are averaged out to give an overview without giving too much attention to other statistical measures such as percentiles or outliers and how they affect the results. It is discussed on WRK's GitHub issue threads that since the WRK tool's primary focus is to measure throughput and not latency. Therefore, the latency statistics can be left as 'extras' and the test can focus on the most useful measurement for this paper, which are requests per second.

Another issue also discussed in various issue threads on the tool's GitHub page is related to how the scripting could affect the performance of the tool itself. It was mentioned that under certain circumstances such as modifying the behavior of a request when it comes back (with Lua scripts) to perhaps calculate something and store values in temporary

variables can indeed affect performance. Therefore, it is important to keep in mind that the only recommended behavior change for Lua scripts is prior to issuing requests. These changes involve keeping track of counter variables or adding body to requests or changing their path.

3.5.2 Reason for choosing WRK over Locust

WRK is a C based benchmarking tool and thus becomes fairly easy to understand for someone who started the journey of programming and computer architectures with this language. However, that's not the only reason for using WRK for benchmarking. It has scripting potential developed in the Lua programming language that is very easy to understand. Also, there are plenty of examples with Lua scripts on the supported GitHub page (WRK, 2020) as well as many explained concerns covered in the previous section.

3.6 High level designs

3.6.1 Design components

Based on what was discussed in previous sections and how benchmarking will help in evaluating the performance of the two architectures in question, the design of the system architecture will involve the following elements:

1. **Benchmarking tool host** - to host and run the WRK benchmarking tool. This is a dedicated virtual machine to ensure the tool has enough logical ports and processing power.
2. **Microservice and FaaS host** - Similar to benchmarking, a dedicated host for adequate memory and processing power. In the case of FaaS this is not available, however the component can be entirely replaced by FaaS equivalent technology.
3. **Load Balancer** - A HTTP load balancing server, fully managed by a cloud provider. It sets up a static HTTP endpoint which is used to forward requests to a microservice host. It's important to have it as any additional microservice hosts can be easily linked and the targets can be swapped over. Very useful as microservice machines get torn down and new private IP addresses are assigned.
4. **Database server** - Since the initial service design from section 3.1 is working with permanent data, and to follow microservice principles, the database server

will act as a permanent data storage. It is required to have it as data is not meant to be stored permanently on microservices.

5. **Cache server** - The service works with permanent data, however this data is also transferred on a frequent basis. A caching server is to offload the stress of the database server and allow for frequently used data to be reused from memory that is accessible in the fastest way.

3.6.2 Microservice high level design

In order to test microservices and FaaS architectures in the most optimal manner, both must be set up in a way where both technologies can be swapped over without changing overall architecture. Figure 3.9 pictures how a microservice will fit into the overall system architecture. Benchmarking tool will be deployed to a virtual machine instance to simulate performance load coming from one or more components throughout the system. It is expected that one machine will issue more load than enough as the WRK tool is very efficient and only limited by the amount of sockets it can open on the machine it is run from. This setup is also compatible with deploying benchmarking tools on multiple machines, as the load balancer helps with spreading the load to multiple machines.

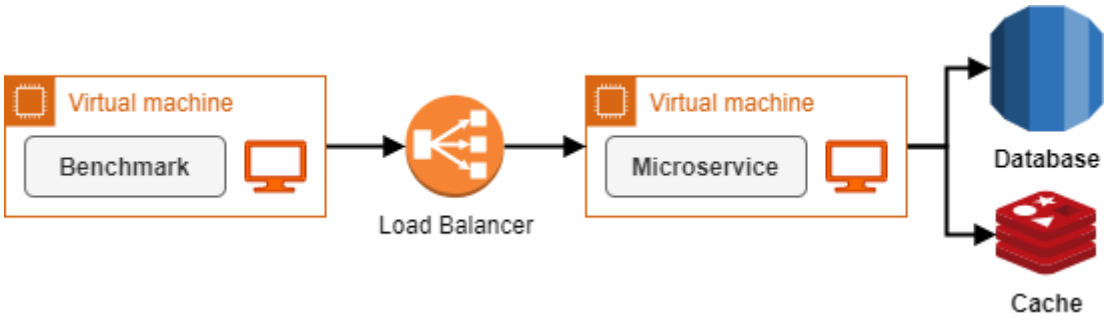

Figure 3.9 High level design for Microservice solution

Deployed benchmarking tools will issue HTTP calls and contact microservices almost directly. Microservice machines need an appropriate load balancer which they can use as a frontline. This will be the only way to call microservice applications. The type of load balancer used for both Microsoft Azure and AWS will be Application Load Balancer, a specially crafted load balancer for HTTP/HTTPS workloads only. This means no other communication protocol can be used with this load balancer. What is good about this type of load balancer is that for both cloud providers, a private and public

resource link is generated. Since this architecture is resembling communication between backend systems, the load balancer will only be configured to use a private resource link or URL for short.

3.6.3 FaaS high level design

As discussed in section 3.1 the FaaS infrastructure can be done in multiple ways. Both cases using singular functions per different HTTP call and a single function for everything will need to be tested. For the sake of keeping the codebase very closely the same, the decision was made to keep all the logic for POST, GET, PUT and DELETE in single function as the logic of the requests is more or less the same (Section 3.8). The advantage of this approach is portability and reusability, which is also a design principle used for microservices as discussed in Chapter 2.

In the case of FaaS, a similar approach will be used as with microservices. However, using a load balancer with a commercial cloud FaaS provider seems redundant and unnecessary. On top of that, in AWS it becomes even more difficult as an AWS Lambda invoked by load balancer waits until the Lambda finishes executing until it does it again. This unfortunately is a limitation in AWS which luckily can be replaced with something else. Another reason for not using load balancer is because it is not possible to control FaaS schedulers or its own internal queueing or load balancing. Therefore for FaaS it will be better to use a simple API Gateway in case of AWS Lambda (Figure 3.10) and in case of Azure - Azure App Service straight away (Figure 3.11).

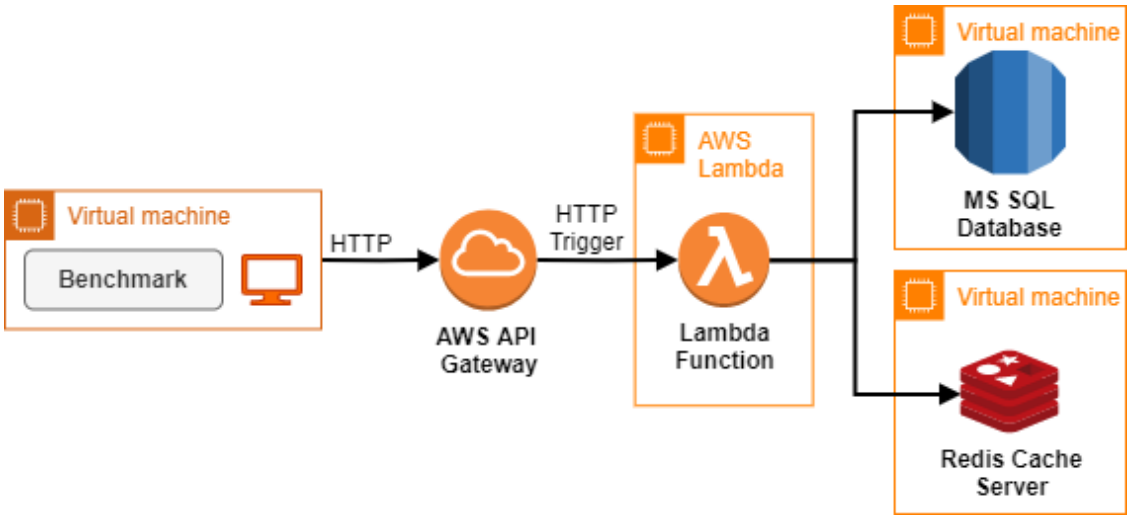

Figure 3.10 High level design for FaaS solution - AWS

An API Gateway in AWS resembles a simple HTTP/HTTPS routing system, configured by the web-based control portal. Configuration of the gateway allows for calling or executing instructions, such as executing a Lambda function when a specific endpoint is called e.g. as originally planned Get request is called, a Function will be called with appropriate parameters. What is good about this approach is that the AWS API Gateway is specially designed for this task and is able to do so, with as many as 10000 executions per second or higher if the architecture is allowed to do so. The only problem with such high throughput could mean a very high level of concurrency invocations in AWS Lambda is limited. However, since the purpose of the test is to see how good the performance of this setup is, any issues will be seen after results are recorded.

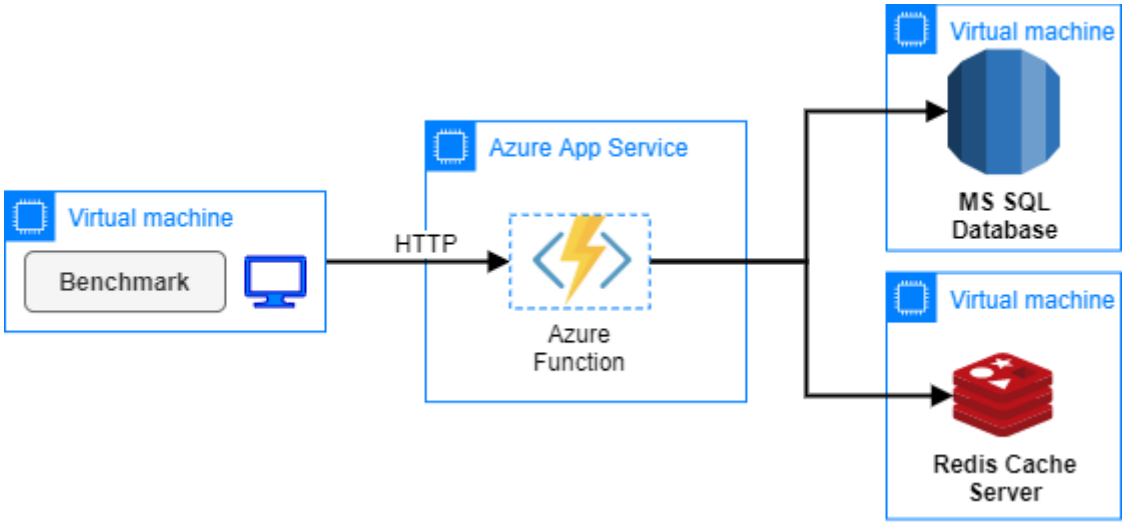

Figure 3.11 High level design for FaaS solution - Azure

With Azure App service the story is slightly more abstracted than in AWS. The load balancing that is happening in Azure Functions is not controllable and Azure Function is leaving the user with a created URL to the resource. This is simulating load balancing which is normally done manually but in Azure - abstracted away, leaving an impression that the FaaS provider is expecting the user to use this service in this specific way. Of course, Azure Functions give more options regarding how Functions are invoked other than HTTP. However, since Azure is already providing all means necessary to contact the FaaS service in the desired way, other means are not important for this project.

3.7 Database and Caching

3.7.1 SQL Database Server

Since both microservices and FaaS applications will convey the same types of requests, they will also both use the same permanent and temporary data storages. In the case of a database, a SQL based, relational database will be used. Primary choice for this matter is Microsoft SQL database. In Microsoft Azure this is supported by default in Azure SQL. However, in AWS this can be achieved by setting up an Amazon Relational Database Service (RDS) database with Microsoft SQL as a supported engine.

Another option considered for this project was MySQL database, also supported by both cloud providers. The performance of the database might be crucial for certain tasks executed by both microservices and FaaS functions. However, these tasks are not database intensive and possibly the only factor affecting these tests is how fast can both architectures access and retrieve the small amount of data they are going to manipulate.

In order to make the database technology behave in exactly the same way across all planned environments it will be important to use the same technology, and if possible same server capabilities regarding virtual CPU and memory amounts. For this purpose, Microsoft SQL will be more than enough. On top of that both Azure and AWS environments provide means to set up exactly the same version of MS SQL servers, which in this case means MS SQL 2019 version.

3.7.2 Caching Server

Out of all open-source to commercial caching options out there, the decision was settled around Redis (Redis, 2020), a high throughput and fully open-source caching framework which serves as a simple key-value caching store. With in-built Redis features such as key-value expiration times, it is possible to set up more realistic testing scenarios where data is cached only for a short period of time before the data has to be retrieved from a permanent data store once again.

Both Microsoft Azure and AWS provide the necessary means to setup cache technologies such as Amazon ElastiCache or Azure Cache. However, since both were specifically designed to work within respective cloud provider environments they might

be limited or set up in a way which is against the networking design. An example of this would be Azure Cache which exposes a public TCP endpoint by default that is accessible from outside of a virtual network. Instead, to avoid a setup with exposed networking it will be better to set up a self-made virtual machine with Redis running as a service. That way the networking is more controllable and a guarantee that the service operates exactly the same way in both cloud environments.

Redis is a purely Linux based service and so, if one would require to setup a Redis cache server, it is necessary to setup a Linux based virtual machine. It is not expected to have a performance difference between a commercial version and a self-setup version. In fact a privately available caching service could only yield better performance on networking level due to shorter address lookup of the caching server. Additionally, considering the caching servers are set up as the same Linux machines it is possible to eliminate potential performance differences since the cloud providers version of the service could enforce the service to be run from a container service or a different operating system.

3.8 Networking and automation

3.8.1 Networking

In section 3.2 it was discussed how preferably only one machine is used to access the infrastructure from outside a virtual network. The so-called ‘jump boxes’ used to access back-end infrastructure are more commonly referred to as bastion machines. A bastion machine is able to access any other component on the given virtual network, similarly to LAN connectivity in every household. This bastion machine will be used to access and retrieve logs, test any virtual machines in proof-of-concept manner and install remotely any other software required to conduct testing on these machines. Figure 3.11 is a visual representation of this idea.

For both microservices and FaaS components will be kept in separate subnets. Both Microsoft Azure and AWS allow creating separate subnets and give control over which can be exposed to the public internet. The bastion virtual machine will be kept in its own subnet which will be exposed to the internet. Data storage, benchmarking tools and subjects of the test will have subnets configured in such a way to avoid any IP address starvation, having enough space to assign multiple addresses where necessary. It is also

important to do that because in case microservices will be added into the load balancer as testing progresses.

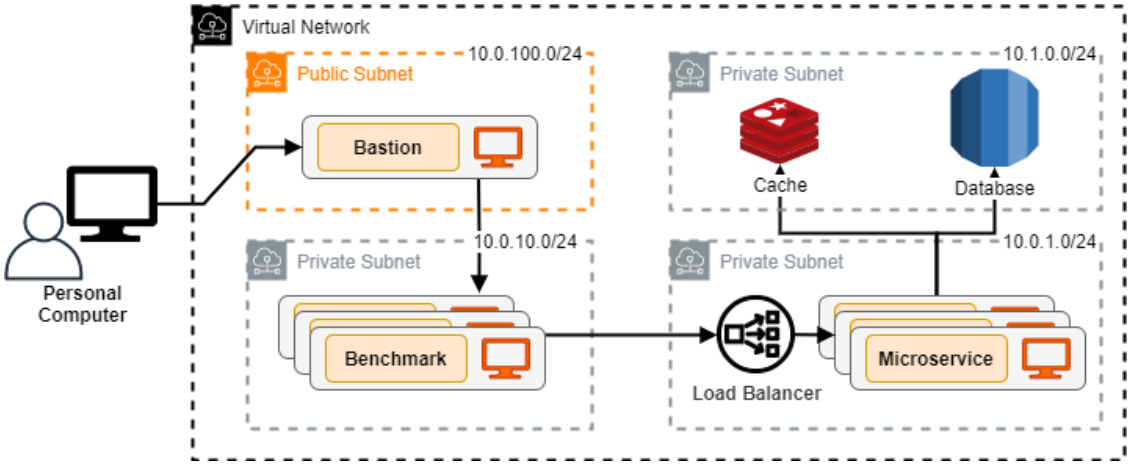

Figure 3.12 Microservice networking

In FaaS adaptation networking will be kept as closely as possible to microservices. Figure 3.12 pictures how API Gateway and FaaS technology can be kept within one subnet. Not to be confused with Internet Gateway which can reside outside of any subnet. To emphasize, FaaS technology does not give any visibility on physical machines and trying to reverse engineer what is in fact happening in both Azure functions and AWS Lambda regarding networking might not be possible. As discussed in chapter 2, the best guess to make is that networking is entirely managed by the FaaS provider itself.

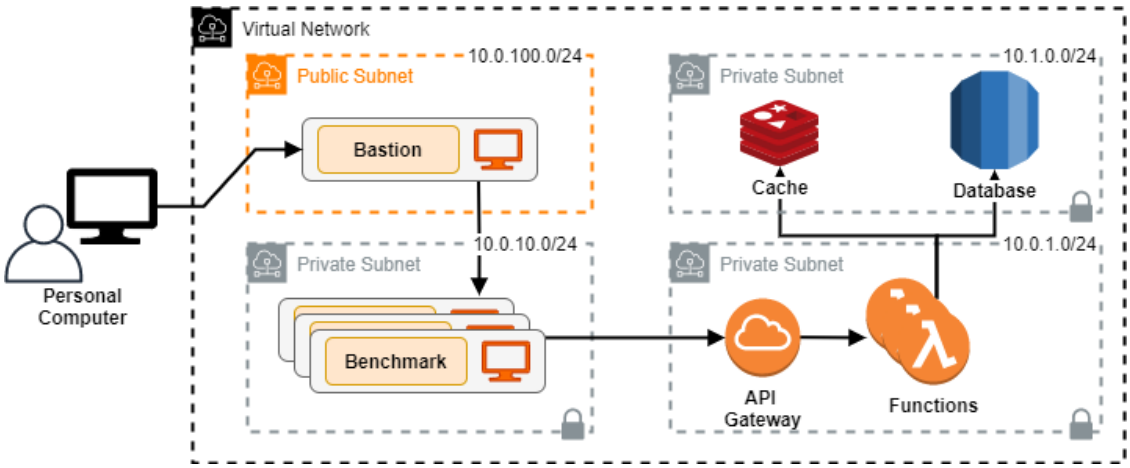

Figure 3.13 FaaS networking

3.8.2 Automation and reusability

To make testing reusable in some sense, an appropriate automation needs to be set up. For this purpose, no third party frameworks will be used as the architecture is not that large. Instead, carefully designed PowerShell scripts will do more than enough to facilitate deploying the code, initializing benchmarking tools and retrieving any logs which virtual machines setup for microservices might create.

PowerShell scripts will make the tests more controllable and reusable. Main communication method with virtual machines setup in private subnets will be SSH. Bastion box is a Windows machine and is accessible through a Remote Desktop Protocol (RDP) connection. Scripts will be executed directly through the bastion box. However, any private IPs setup for the virtual machines inside private subnets will need to be configured inside the windows PowerShell script either as a parameter or hardcoded.

3.9 Testing plan & benchmarking workloads

With the use of the WRK tool, this paper proposes a PCP (Piertraszewski's Cost-Performance) benchmarking suite in order to correctly evaluate the performance of the architectures in question. The main goal of PCP benchmarking suite is to fully test the ability of two 'safest' HTTP calls of GET and PUT, which also introduce enough complexity for the service to have noticeable performance differences during execution.

PCP proposes specific values used for WRK, which are:

- 100 connections
- 10 threads
- Duration of 300 seconds
- Additional Lua scripts for changing URL parameters or body

Number of connections set to 100 will make sure that the 10 threads have equal amounts of channels they can issue HTTP requests from. It will also make sure not go over potential networking limits of the machine. This number could be higher, however, the assumption here is that the benchmarking machines will have low computing power to utilize networking to its full capacity. Optimally, the test duration could be anything

between 2 to 10 minutes. Five minutes here are projected to generate enough results and costs to give a good overview of the performance capability of the service in question.

Lua scripts will make sure the data that is used for the benchmark gets equally distributed among all requests. For example, a Lua script will change the body parameters that are used to update records using PUT requests or change the path of the URL when issuing GET requests.

An example of a GET Lua script used with WRK for this benchmark looks as follows:

```
local maxAccountNumber = 10000
counter = 0
request = function()
       counter = (counter + 1) % maxAccountNumber
       wrk.method = "GET"
       wrk.path = "/api/account/" .. counter
       return wrk.format()
end
```

WRK tool executes this script separately on each thread as specified with thread number. The account number used with the script is almost cosmetic, however 10000 seems like a good number as in conjunction with thread number and available connections the threads produced by the WRK tool will have plenty of time to go through the counter, and then start from 0 again.

3.9.1 Workloads

PCP benchmarking suite suggests the use of customized workloads to test the performance of the web application designed using the two architectures.

Workload	Operations	Description
A - Read heavy	Cache Read: 80-90% DB Read: 5-10% Cache save: 5-10%	Data retrieval from primary store and caching
B - Update heavy	DB Update: 50% Cache Delete: 50%	Data update in primary store and cache key delete
C - Insert only	DB Insert: 100%	Inserting new data
D - Delete only	DB Delete: 50% Cache Delete: 50%	Deleting existing data and clearing cache

Table 3.1 PCP Workloads

Workload A is a primary choice for evaluating the performance. By running a five minute test, most operations are projected as cache reads while the rest grab the data from primary source and cache them.

Workload B is equally updating both the database and the cache, with the exception of deleting the key from the caching server.

Workload C is designed to purely insert data into a database. It was designed as a workload mainly to test capability of the primary data store to insert data. It's not going to be as useful when evaluating performance and costs but will serve as a preparation for other workloads.

Workload D is deleting the data inserted by workload C, and also clears cache keys to reset the cache. It is also not as useful as other workloads however, with automated testing, the workload will not only reset the environments but also test the capabilities of data stores and cache to delete records.

Using PCP benchmarking suite, the designed workloads will be tested against the two architectures. Next sections will go over execution details of these workloads, giving a good mid-level overview of their inner workings.

3.9.2 Workloads A, B - GET and PUT requests

Most commonly used out of all. The GET request can be considered as a 'go to' scenario as resources are often requested very frequently for further processing. This also means that potentially a lot of data is going to be transferred between servers. However, the amount of data is not as important in this scenario, because the variation of the amount of data could often be related to media sharing and streaming. Using the preliminary design from section **3.2** a more important measure will be consistency. To make testing consistent the data transferred between servers must be more or less the same, a couple of bytes won't make too much difference.

The behavior of GET request can be described with following pseudocode:

```
START
```

```
        VARIABLE record = CALL.CACHE_GET()
        IF record IS NULL
                record = CALL.DATABASE_GET()
        END IF
        IF record IS NOT NULL
                CALL.CACHE_SET(record)
        END IF
        RETURN record
END
```

The idea of caching here is to find a potential point when the performance or requests per second value is improving because the records are now loaded from a more lightweight source. However, this could be only seen if using large records and a heavily populated database and therefore possibly very expensive to test in the long run. The logic will be preserved to promote good and proper practices of microservices.

With PUT requests there is a similar story, however the data transfer direction is reversed, the calling benchmarking server is transferring the data to the tested web server. The execution of the request can be then depicted as follows:

```
START
        VARIABLE result = CALL.DATABASE_MODIFY(record)
        IF result IS TRUE
                CALL.CACHE_DELETE(record)
        END IF
        RETURN result
END
```

Every time the record is updated by the service, it's cached value must be cleared to avoid any data inconsistency. The returning value can be used as a way to indicate whether the request is a success with status code of 200, or status code of 4XX to indicate potential failure.

To measure the performance of GET and PUT requests the tests need some data prior to conducting the tests. Without the data set prior to these tests there is a risk that exception behaviors will render results unreliable, because the test is not focusing on performance of exception handling.

3.9.3 Workloads C, D - POST and DELETE requests

Tests which input and delete data will be used as a starting and ending point of the test. However, such tests are hard to manage. Especially with WRK, because of the multithreading nature of the tool. If each thread in WRK would do the exact same job, the test could cause a lot of inconsistencies since lots of duplicated data would try to be inserted into the database. This in turn would cause the microservice or the function to throw a lot of errors and again, the purpose of this stress test is not to check how exception handling is performing. Another good reason to avoid such scenarios is the cost associated with running functions in AWS. An increased amount of time for processing caused by exceptions would make it difficult to accurately calculate associated costs with running such a test.

Behavior of POST requests is depicted as follows:

```
START
	VARIABLE result = CALL.DATABASE_CREATE(record)
	RETURN result
END
```

With POST type requests no caching needs to happen as the record is being created for the first time. When a GET request is issued for the newly created record, the data is saved to the cache so it no longer is requested from the database.

With DELETE type requests there is a similar story. The database is cleared out of the existing data record after which the cache needs to be cleared from the same record too.

```
START
	VARIABLE result = CALL.DATABASE_DELETE(record)
	IF result IS TRUE
		CALL.CACHE_DELETE(record)
	END IF
	RETURN result
END
```

Both PUT and DELETE requests should be executed in a slow and separated manner. This will ensure that data creation and deletion is happening as expected and as discussed before, the issue with exception handling is avoided.

3.10 Design summary

The overall idea of the experiment became much easier to understand once benchmarking keyword was used. The experiment will focus on benchmarking specific workloads and inspect how they perform under specific circumstances. The test utilizes a WRK benchmarking tool using four request types of POST, PUT, GET and DELETE issuing these requests to a service or function which responds with behaviors described in previous sections. With carefully designed Lua scripts, the WRK benchmarking tool will take care of stress testing the environments. Judging from designed workloads, workloads A and B provide enough complexity and are durable enough to serve as a good basis for evaluation.

Database servers for all environments will be equally the same, meaning that a separate MS SQL server will be installed on machines with the same CPU and RAM, also giving full access to the host machine via RDP or SSH connection.

Caching servers for all environments will also be equally the same. Linux machines with Redis server installed from the official repository on each machine. Also, the machine will be fully accessible over SSH.

Since, the design of the experiment has a very repetitive nature it will be worth investing time into developing any means of automating the deployment and restarting of the environment. It will also be worth automating collecting logs and necessary metrics generated by the test to further validate metrics coming out of WRK benchmarking tool. For all automation necessary PowerShell scripts will be more than enough, they will also set a footprint for anyone who would like to consider doing similar projects.

In the next section, some implementation details are discussed and presented. Some implementation differences for microservices and FaaS architectures are covered and benchmarking configurations are discussed. Additionally, some automation details including its implementation are also covered. Lastly, the next section also goes over gathered results and conducts evaluation upon them.

4 Results, Evaluation and Discussion

This chapter describes how the benchmark from Chapter 3 was developed and executed. For both microservices and FaaS were designed to run in two major cloud platforms namely AWS and Azure. The benchmark consists of 4 workloads. Each of these were run and the results were gathered and analyzed.

This chapter also describes the system configurations prior to executions and the challenges faced. It describes some differences between cloud solutions as well as some problems faced while trying to maintain the same design of the test throughout the development. It also includes some of the results generated throughout the testing phase as well as all high-level DevOp configurations used across cloud environments.

4.1 Runtime environment and software used

4.1.1 Runtime versions

At the time of the development the most recent version of Microsoft's .Net Core long term support (LTS) version of 2.1 was 2.1.805. Ultimately, this version was used in development of microservices. In both Azure and AWS versions of the function a version of 3.1 was used as it was the latest one supported. Version 3.1 version of .Net Core has breaking changes in its MVC framework which were minor and not a concern for functions as MVC framework was not used for functions. Despite the runtime difference the list of changes provided by Microsoft between versions does not list anything worth mentioning as version 3.1 only includes a lot of extras which aren't used in any of the developed solutions.

As an important note, at the time this solution was developed (2012) the LTS version of .Net Core 2.1 projected its end of life on August 21, 2021.

4.1.2 Development software used

Both developments of microservice and function solutions were achieved partially with Visual Studio Code and polished in Visual Studio 2019 (Professional).

To help with development of a Lambda function, AWS provides project templates installed with an extension for Visual Studio available on the main website of AWS (AWS, 2020). Installed project templates include simple AWS Lambda project templates for various types of integrations, such as AWS Kinesis, AWS SQS (Simple Queue Service) or AWS API Gateway which is exactly what is needed for this project. It was also worthwhile to check out the documentation for Lambda integration with API Gateway also provided on the main AWS documentation site. It provides more details as to why the project template is set up in such a way as well as more useful information of possible request values and how to unit test everything locally.

Microsoft Azure also provides project templates for building Azure Functions. As opposed to an officially provided extension, Azure does it directly in Visual Studio installer. Similarly, to AWS Lambda, these templates also include different integration types and provide an easy way to set up a project including a local unit test.

4.1.3 Scripting software used

For automation purposes, many PowerShell scripts were developed in Visual Studio Code and Windows PowerShell ISE which comes with Windows 10 operating system. With a Windows PowerShell console, most automation scripts were tested out on a local machine before copying scripts on to Bastion machines.

Lua scripts specific to WRK benchmarking tool were also developed in Visual Studio Code but were officially tested in mock test environment setup in Azure cloud.

4.2 Implementation consistency

4.2.1 Dependency Injection

When writing code, there may be a time where developed methods or classes require parameters to be more organized and properly instantiated. As the code evolves more parameters and complex objects need to be parsed around and perhaps instantiated in a more complex way, with interfaces, inheritance and abstractions. It may also come into a situation where only one instance of a specific class is desired for the lifespan of the application or a specific scope such as a web request. In this situation a developer would

typically think of a way to intelligently handle creation of objects and control how many are instantiated and how they are requested.

A modern solution to instantiating classes across the application is called dependency injection. With dependency injection, objects are created with 'dependencies' they need, by 'injecting' required objects or values directly from a dependency resolver.

```
public AccountController(ILogger<AccountController> logger, IDatabaseService dbService,
    ICacheService cacheService)
{
    _logger = logger;
    _dbService = dbService;
    _cacheService = cacheService;
}
```

Figure 4.1 Controller constructor

Figure 4.1 represents the main constructor method of AccountController class. It accepts a couple of interfaces used within other methods. In order for these interfaces to be injected into this class, they must be registered somewhere within the code. In the case of this project, they are registered within a **Startup.cs** class with ConfigureServices() method.

```
public void ConfigureServices(IServiceCollection services)
{
    try
    {
        services.Configure<KestrelServerOptions>(_config.GetSection( key: "Kestrel"));
        services.Configure<DatabaseConfiguration>(_config.GetSection( key: "Database"));
        services.Configure<CacheConfiguration>(_config.GetSection( key: "Cache"));

        services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_1);
        services.AddDbService();
        services.AddCacheService();
    }
    catch (Exception e)
    {
        _logger.LogError(e, message: "Failed to Configure Services");
        throw;
    }
}
```

Figure 4.2 Configuring services

Microservices project is using Microsoft.Aspnetcore.App package and with it, access to Microsoft's dependency injection classes and extensions. The Startup class is a little special, it contains certain methods such as ConfigureServices() which are required by the AspNetCore framework in order to set up the web application. Figure 4.2 contains

an example of that method. It shows how certain classes are registered as configurations which are used within the code and some custom extensions to register more services, e.g. AddDbService(), AddCacheService(). The method also registers main mvc classes with the AddMvc() method which will grant all the basic features of the mvc framework from the AspNetCore package.

The FaaS implementations follow a similar implementation theme with dependency injection which are covered in section 4.4.

4.2.2 HTTP Responses

When requests are completed, they need to give an appropriate response code back to the caller. In this case with WRK it is known that the only response code accepted, marking the requests as ‘completed’ will be any code with 20X, where X can be any number. Any other response code will immediately mark the request as ‘failed’ and simply not count it towards the success count.

For the purpose of this test, a 200 response code is used for every request that ran to completion. If a request has failed anywhere in the code and the failure is handled, then a 409 response code is used. Should any other failures occur, the mvc framework will return 500 response code containing an exception message. The nature and the exact code indicating failure does not matter a lot in the context of this test as the WRK benchmarking tool will treat all failing codes equally.

4.2.3 Common libraries and packages

Apart from the big library of Microsoft.AspNetCore.App required by microservice to set up a web server most libraries are shared, and the code remains the same across implementations.

- Dapper - to connect and execute SQL queries on the database.
- StackExchange.Redis - to connect and execute queries on Redis cache server.
- Microsoft.Extensions.Logging - to create lightweight logs with built in extensions to configure with existing dependency injection.
- Newtonsoft.Json - to convert objects into readable and properly formatted text.

4.2.4 Health checks

In order to pre-test all implementations they need a way to be called without creating unnecessary errors. Introducing a health-check call can help with this issue and it also has a different purpose.

```
[HttpGet( template: "")]
0 references | Mateusz, 26 days ago | 1 author, 1 change | 0 requests | 0 exceptions
public async Task<IActionResult> Get(CancellationToken token)
{
    return await Task.FromResult(Ok());
}
```

Figure 4.3 Health check method

When load balancing is set up with many applications, they normally need to be checked periodically by the load balancing server in order for it to know if the application is still available and responsive. In cases where the application is no longer available, the load balancer will mark this specific application instance as unavailable and ignore it with future requests. To inform the load balancer if the application is still available, they contain a very basic request that simply returns 200 or any other code back as soon as it's called, just like in Figure 4.3. The frequency of such calls is configurable like it is in Application Load Balancer in both Azure and AWS.

4.2.5 Time measurement

In order to record specific metrics for measuring how long certain operations took within the code it is important to record metrics in an efficient way.

```
public static class Invoker
{
    11 references | Mateusz, 15 days ago | 1 author, 2 changes | 0 exceptions
    public static async Task<double> Invoke(Func<Task> action)
    {
        var stopwatch = new Stopwatch();
        stopwatch.Start();
        await action.Invoke().ConfigureAwait(false);
        stopwatch.Stop();
        return stopwatch.Elapsed.TotalMilliseconds;
    }
}
```

Figure 4.4 Stopwatch time measurement

Designed method shown in Figure 4.4 is used to record metrics in a somewhat generic way. The method takes any asynchronous action and executes it in a new context. Created stopwatch records the time during the action and the accurate timestamp is returned in the form of milliseconds.

What is good about this measurement is that the value returned by 'Elapsed' is a timestamp and the millisecond value is very accurate, containing trailing digits indicating micro and even nano seconds. Stopwatch contains another few values, including a direct millisecond value. The only problem with this direct millisecond value is that it is rounded. A rounded value of millisecond could be enough in a more business environment where a whole request operation request time is recorded. However, in the case of benchmarking, and especially with this test, a more accurate value is required and luckily - provided by .Net Core framework.

4.3 Implementing in-code services

4.3.1 Microservice composition and hosting

In Section 4.2.1 it was mentioned how Microservices utilize the Microsoft.Aspnetcore.App library to set up the basis of the infrastructure. The way Model View Controller (MVC) framework is setup in AspNetCore can be tricky if documentation has not been studied before. For example, there is a specific reason as to why certain classes absolutely need to have words like 'Model' or 'Controller' in their names or be part of a folder with such a name. During creation of the hosting service, the scanning for the controllers and other types of classes used in AspNetCore is taking place by a process called 'reflection'. A type of programmatic scanning and instantiation of objects. In case of implementation in this microservice it was decided to keep every HTTP, method implemented within a class called AccountController.

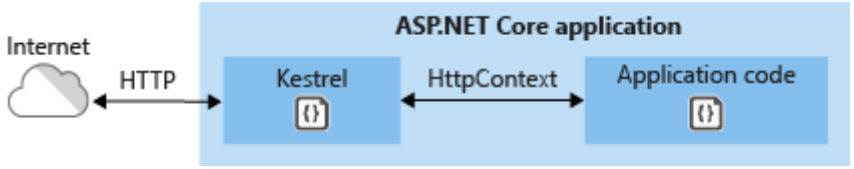

Figure 4.5 Kestrel server host

The web server is set up using Kestrel as the main host. What that means is that the application is opening a port directly on a server and the application handles all network traffic for that port, like it is shown in Figure 4.5. The choice of port was left at 4000, just to make it safe and away from other services. Why port 80 was not used is because port 80 is usually reserved by the system for other means. Since it's not necessary, port 80 was not overridden on the hosting operating system.

Another thing to mention with hosting microservices is how they are actually run on a server without the user present on the machine. To do so, the application was designed to run as a Windows Service.

```
var isService = !(Debugger.IsAttached || args.Contains("--console"));
if (isService)
{
    logger.LogWarning(message: "Running as Service");
    host.RunAsCustomService();
}
else
{
    logger.LogWarning(message: "Running Interactive");
    host.Run();
}
```

Figure 4.6 Microservice as Windows service

One problem encountered with setting up an application as a Windows service is having the code figure out if current execution is intractable by the user or not. One way of doing so would be using Environment.UserInteractable() method from System. The problem with this approach is that it doesn't always work, especially with self-contained applications such as this project. Therefore, the application entry point was supported with the code from Figure 4.6 to make sure no situation occurs. Setting up the application as a service happens in a small extension compressed into a RunAsCustomService() method which simply encapsulates the hosting service into a WebHostService from AspNetCore library.

4.3.2 Database Service

Contacting the database is a significant job, and thus, requires few dependencies. The responsibilities of connecting to databases and invoking SQL statements deserves its

own space, not only for organizational purposes but also so that dependency injection can fit nicely into the picture.

```
public interface IDatabaseService
{
    3 references | Mateusz, 15 days ago | 1 author, 1 change | 0 exceptions
    Task<AccountData> GetAccount(int accountId, CallStats stats);
    3 references | Mateusz, 15 days ago | 1 author, 1 change | 0 exceptions
    Task<bool> CreateAccount(AccountData data, CallStats stats);
    3 references | Mateusz, 15 days ago | 1 author, 1 change | 0 exceptions
    Task<bool> ModifyAccount(AccountData data, CallStats stats);
    3 references | Mateusz, 15 days ago | 1 author, 1 change | 0 exceptions
    Task<bool> DeleteAccount(int accountId, CallStats stats);
}
```

Figure 4.7 Database service interface

Figure 4.7 shows how the database service class was designed to work within the application. All methods accept a parameter which links to the correct object stored in the database, and also an object for keeping track of statistics related to a specific chain of calls.

At core, the SQL statements are kept as constants as they do not change. If for example a database server changed during development with a different technology than Microsoft SQL Server, the code would not have to change at all.

```
public async Task<AccountData> GetAccount(int accountId, CallStats stats)
{
    try
    {
        IEnumerable<AccountData> result = null;
        var elapsed :double = await Invoker.Invoke(async () =>
        {
            using (var connection = new SqlConnection(_connectionString))
            {
                result = await connection.QueryAsync<AccountData>(
                    SqlGetAccount, param: new {AccountId = accountId});
            }
        });
        stats.AddStat(StatType.DatabaseGet, elapsed);
        return result.FirstOrDefault();
    }
    catch (Exception e)
    {
        _logger.LogError(e, message: $"{nameof(GetAccount)}() Exception while getting account");
        throw;
    }
}
```

Figure 4.8 Database Get implementation

As for implementation of each method, the GetAccount() will serve as an example as shown in Figure 4.8. The important part of this method is how the connection with the database is happening in its own dedicated block of 'using' statement. What this means essentially is that the connection itself is new and instantiated pretty much with every request coming to the web server. An advantage of that is the lack of concern over connection manipulation, it is simply created, a call is executed and then the connection is immediately terminated.

4.3.3 Cache Service

Caching is another important job the service has to perform. When it comes to caching, there isn't any specific standard since the technology choice is quite vast. Caching can be in-memory, service based, or even remote. In case of this implementation caching is done remotely, with Redis server as technology choice. Redis is a very advanced key-value store. Meaning the data is literally kept in key-value format, there are no structures or relations and Redis server only has some limited support for lists which are not used in this project.

```
public interface ICacheService
{
    7 references | Mateusz, 41 days ago | 1 author, 1 change | 0 exceptions
    bool Enabled { get; }
    2 references | Mateusz, 15 days ago | 1 author, 1 change | 0 exceptions
    Task<AccountData> GetAccount(int accountId, CallStats stats);
    2 references | Mateusz, 15 days ago | 1 author, 1 change | 0 exceptions
    Task<bool> SaveAccount(AccountData data, CallStats stats);
    3 references | Mateusz, 15 days ago | 1 author, 1 change | 0 exceptions
    Task<bool> ClearAccount(int accountId, CallStats stats);
}
```

Figure 4.9 Caching service interface

In Figure 4.9 the interface for the caching server is presented. It contains similar methods to database service implementation mentioned in the previous section. It doesn't have a method for modifying data since in key-value stores such as Redis it doesn't make sense as the value of the key is simply overwritten.

```
public async Task<AccountData> GetAccount(int accountId, CallStats stats)
{
    try
    {
        RedisValue result = RedisValue.Null;
        var elapsed :double = await Invoker.Invoke(async () =>
        {
            result = await _multiplexer.GetDatabase().StringGetAsync( key: accountId.ToString());
        });
        stats.AddStat(StatType.CacheGet, elapsed);

        if (result.HasValue)
        {
            return JsonConvert.DeserializeObject<AccountData>(result);
        }
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, message: "Error while saving to cache");
    }
    return null;
}
```

Figure 4.10 Caching Get implementation

Figure 4.10 contains implementation for the GetAccount() method. Unlike connecting to sql database, the connection is maintained by another interface which is injected into the CacheService class. When executing operations such as StringGet, this interface will open a new connection and execute a TCP call to the server. Resulting object, if not empty, is then deserialized with the previously mentioned Newtonsoft.Json package. Relevant statistics with how long this process took is also recorded.

4.4 Lambda function differences

4.4.1 Dependency injection and entry point

A fundamental difference between the microservice implementation and any FaaS function implementation is the lack of framework. A function designed as a lambda is as basic as a console application. The only information AWS Lambda needs to execute a function is the path to the assembly and method which act as the entry point, very similar to how a Main(args[]) function works in most programming languages.

With this difference the implementation of the Lambda needs to make sure that dependency injection is also in place like it is in case of a microservice. In which case the dependency injection is almost identical as described previously in section 4.2.1.

```
public async Task<APIGatewayProxyResponse> Run(APIGatewayProxyRequest request,
    ILambdaContext context)
{
```

Figure 4.11 Lambda entry function header

This function is invoked by calling Run() method as shown in Figure 4.11. Each AWS Lambda takes 2 parameters. First parameter depends on the type of integration chosen, although by default it is a string object that is simply serialized into something different. Second parameter is a lambda context that is always supplied by AWS Lambda. It contains some basic information regarding the function itself, such as runtime, version, or the unique id of execution.

4.4.2 Reason for using API Gateway

Since this Lambda is going to be invoked with a HTTP call, it needs to be integrated with another AWS service such as API Gateway or Application Load Balancer (ALB). For the purpose of this test an API Gateway had to be chosen. As it stands currently, the ALB integration is new and has not matured yet to be somewhat useful. During initial testing of the Lambda designed in this project an ALB was initially used to test how it performs. The results were unexpectedly bad. It seems that when the ALB receives a request which needs to be forwarded to a Lambda, it forwards the request and waits until it finishes before executing again. Because of that, AWS Lambda cannot be used to its full potential as it is now only executed one at a time, synchronously. With API Gateway it is possible to call Lambda with a much greater level of concurrency. Executing hundreds at a time, perfect for designed benchmark.

4.5 Azure function differences

4.5.1 Dependency injection and entry point

Azure functions follow a similar story to AWS Lambda. No specific frameworks and mainly bearing differences in execution point.

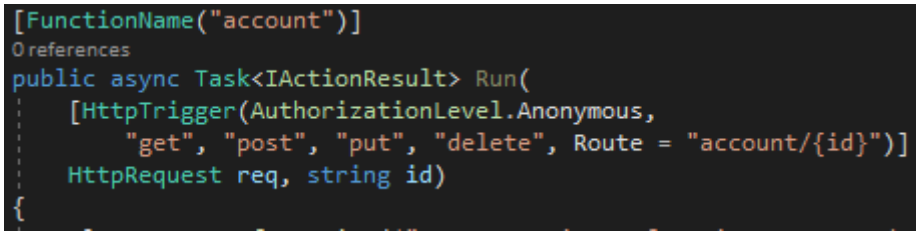

Figure 4.12 Azure function entry point

With Azure Function a special Tag is used to indicate which specific method is meant to be run as an entry point (Figure 4.12). What is different here from the AWS Lambda are the parameters accepted. This function is meant to be triggered with a HTTP call, similarly to Lambda. The parameters for triggering the function have more control over how the HTTP call is handled. They give options for which exact HTTP method to handle as well as additional routes, which is done similarly to standard AspNetCore projects. On top of that the resulting object is exactly the same as the one returned by microservice.

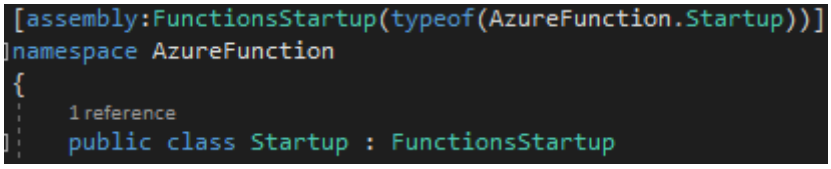

Figure 4.13 Azure function dependency injection

Azure Functions have one more feature that could make them a little better than AWS Lambda. By inheriting a special class 'FunctionStartup' (Figure 4.13), it is possible to tell Azure Function about a method which handles dependency injection and other registrations. Another ability of adopting this class is the ability to grab environment variables which are handled by Azure portal directly, giving the possibility of changing variables on the go.

4.5.2 Function plan and other integrations

One of the differences in how Azure Functions are used as opposed to AWS Lambda in this test is the lack of other integrations necessary to start calling the Function. Azure Function when created and deployed successfully produces a HTTP and HTTPS links for the user to call directly. That means no other integrations are necessary for the WRK benchmark to start calling the FaaS technology directly. What is important to note is that

configuring this function to be called from private networks only requires the choice of a premium plan.

Azure Functions offer premium plans not only to allow private network integration but also to choose how powerful the machine is running this particular function. The premium plan goes up in tiers and each step is introducing more CPU cores and RAM than the previous. This resembles exactly how a normal virtual machine instance is chosen, created and scaled. This creates a fundamental difference between AWS and Azure FaaS technologies. The availability or the choice of how intensely a user wants to use the technology. Azure does it with machine specifications directly where in AWS Lambda this is controlled by a concurrency setting. There are no other settings and global concurrency for AWS Lambda as of today can be up to 900 with 100 remaining in reserve. This difference will be reflected once the results from the benchmark are gathered and analysed.

4.6 Automation and calculating metrics

Automation of this test was achieved with writing PowerShell scripts which were executed from Bastion boxes as explained in Chapter 3. The automation covered initial deployment of microservices, initiating the benchmark and gathering logs from microservice machines. To execute remote commands, a third party PowerShell library called Posh-SSH was used to issue commands on remote servers.

4.6.1 Microservice deployment

Deployment of microservice was achieved in two phases. First phase is to upload a zipped package with the pre-compiled microservice code to the selected server. Along with the zipped package a bunch of utility scripts are also uploaded. The utility scripts are for restarting the environment, deploying the service from a zipped package and preparing the settings file which the application reads from.

The script block to upload the service is shown in Figure 4.14. It has plenty of logging out to the console since the script is run from a PowerShell command line. What is important to note is that even if one line of code in PowerShell script fails or produces an error the script still continues and invokes other lines unless the error is expected by

the developer and appropriate code is written. However, in this case expecting errors is mildly important as the code is trivial enough. Errors of SSH nature usually mean that a connection is not permitted exclusively from the DevOp perspective, in which case one would need to make sure the communication via port 22 is allowed from either AWS or Azure consoles.

```
# Establish the SFTP connection
Write-Output("Uploading package to selected servers...")
foreach($Ip in $ServerIps){
    $SFTPSession = New-SFTPSession -ComputerName $Ip -Credential $Credentials -AcceptKey:$true
    if(!($SFTPSession).SessionId) {
        # Upload files to the SFTP path
        Write-Output("Uploading $PackageName to Server $Ip")
        Set-SFTPFile -SessionId ($SFTPSession).SessionId -LocalFile $PackageName -RemotePath $Re
        foreach($Script in $Scripts){
            Write-Output("Uploading $Script to Server $Ip")
            Set-SFTPFile -SessionId ($SFTPSession).SessionId -LocalFile $Script -RemotePath $Rem
        }
        #Disconnect SFTP Session
        Write-Output("Removing SFTP Session with $Ip")
        Remove-SFTPSession -SessionId ($SFTPSession).SessionId

        Write-Output("Package uploaded successfully..")
    }
    else {
        Write-Output("Establishing sftp connection failed...")
    }
}
```

Figure 4.14 Microservice upload script

Invoking the script remotely is done similarly to how files are uploaded. This is shown in Figure 4.15 which follows a similar pattern. The script block initiates a new SSH connection with every server configured and tries to execute the previously uploaded PowerShell script. Additionally, the output of the remote script to the console is captured and displayed on the host console of the Bastion box.

```
$SshSession = New-SSHSession -ComputerName $Ip -Credential $Credentials -AcceptKey:$true
if(!($SshSession).SessionId) {
    Write-Output("Executing Deploy script on $Ip...")
    $result = Invoke-SSHCommand -SessionId ($SshSession).SessionId -Command "powershell .\D
    $formatted = $result.Output|ForEach-Object {"$Ip> " + $_}
    Write-Output($formatted)
```

Figure 4.15 Microservice deploy script

4.6.2 Benchmark execution

Benchmarking is executed in a similar fashion to how microservices are contacted over SSH. Figure 4.16 shows how benchmarking is initially set up from a master script which orchestrates the whole test.

```
$BenchmarkScript = {
    param(
        [string]$Ip,
        [string]$RemoteUser,
        [string]$RemotePass,
        [string]$TestType,
        [string]$BaseAddress,
        [int]$Duration = 100
    )
    $Wrk = "./wrk-4.1.0/wrk"
    $Command = ""
    switch ($TestType) {
        "Basic" { $Command = $Wrk + " -t1 -c1 -d" + $Duration + "s " + $BaseAddress }
        "GET" { $Command = $Wrk + " -t10 -c100 -d" + $Duration + "s -s ./wrk-4.1.0/Benchmark_GET.lua " + $BaseAddress }
        "POST" { $Command = $Wrk + " -t1 -c1 -d" + $Duration + "s -s ./wrk-4.1.0/Benchmark_POST.lua " + $BaseAddress }
        "PUT" { $Command = $Wrk + " -t10 -c100 -d" + $Duration + "s -s ./wrk-4.1.0/Benchmark_PUT.lua " + $BaseAddress }
        "DELETE" { $Command = $Wrk + " -t1 -c1 -d" + $Duration + "s -s ./wrk-4.1.0/Benchmark_DELETE.lua " + $BaseAddress
    }
```

Figure 4.16 Benchmarking setup script

The prepared command from Figure 4.16 is used as a command parameter to be issued via SSH connection to the benchmarking server. The process is similar to invoking scripts for microservices in the previous section.

4.6.3 Retrieving logs

In order to do analysis, the logs generated inside the code and explained before need to be gathered from one or more servers. The process is similar to uploading code to microservices, except instead of uploading files, they are downloaded from the server. This is shown in Figure 4.17. In addition to downloading logs they are also renamed locally to properly identify which test the logs correspond to.

```
Write-Output("Downloading logs from $Ip")
$LocalPath = "Logs_" + $Ip
New-Item -ItemType Directory -Force -Path $LocalPath
Get-SFTPFile -SessionId ($SFTPSession).SessionId -Remotefile $RemotePath -Local
$LocalFile = ".\" + $LocalPath + "\" + $FileName
Write-Output("Renaming file $LocalFile to $NewFileName...")
Rename-Item -Path $LocalFile -NewName $NewFileName -Force
```

Figure 4.17 Benchmarking setup script

4.7 Environment configurations

4.7.1 Machine specifications

In chapter 3 a high-level overview was given regarding different set ups depending on the environment. With microservices deployed on individual virtual machines to FaaS technology and necessary integrations, such as API Gateway with AWS Lambda.

Purpose	Environment	Machine plan	# of Instances	vCPU(s)	RAM
Microservice	AWS	t2.micro	1	1	1
Microservice	Azure	B1s	1	1	1
Benchmark	AWS	t2.micro	1	1	1
Benchmark	Azure	B1s	1	1	1
Database	AWS	c5.large	1	2	4
Database	Azure	B2s	1	2	4
Lambda	AWS	N/A	Unlimited (900)	N/A	N/A
Function	Azure	EP1	N/A	210 (ACU)	3.5
Function	Azure	EP2	N/A	420 (ACU)	7
Function	Azure	EP3	N/A	840 (ACU)	14

Table 4.1 Environment machines setup

Table 4.1 lists different instances that were used throughout testing. For microservices, only 1 instance per environment was chosen. In case of both AWS Lambda and Azure Function it is not possible to tell how many machines were used. One may notice the difficulty in explaining the AWS Lambda capability, and the short explanation is trade-secret. The only leading piece of information around AWS Lambda capability is maximum concurrency offered by AWS which is a value of 1000 - where 100 is reserved for dynamic invocations across the account.

In the case of Azure Functions, it is not listed clearly by Microsoft whether the premium plans used are operating on one machine only or more. What was listed however are the specifications for each plan, which can leave a speculation that developed Azure Function could be in fact running on one machine only.

4.7.2 WRK benchmark configuration

In addition to the aforementioned configurations, benchmarking setup can be seen in table 4.2. It lists how the WRK benchmarking tool was configured for all request types.

Test	Threads	Connections	Duration
GET	10	100	300 seconds (5 minutes)
PUT	10	100	300 seconds (5 minutes)
POST	1	1	300 seconds (5 minutes)
DELETE	1	1	300 seconds (5 minutes)

Table 4.2 WRK benchmark configurations

Both POST and DELETE request types are not important in the context of this test. Reason being that both of these requests would require more scripting than necessary in order to test properly. For example, since WRK only knows to constantly publish requests within the time (Duration) that was supplied with the command, each thread executing scripts would need extra time to process whether a request should be sent or not. For POST request type this means making sure no exception intended requests are made since the data from POST requests is meant to be inserted only once.

For both GET and PUT test an optimal number of 10 threads and 100 connections was chosen as a ‘safe bet’ not to overextend the capability of the benchmarking machine.

4.8 Results and post analysis

Based on the configurations described in the previous section the tests were conducted on each environment. After each single test the generated logs were gathered using means available and computed locally. Each single test was carefully conducted in the same availability zone of each virtual network.

4.8.1 Workload A - Performance per environment

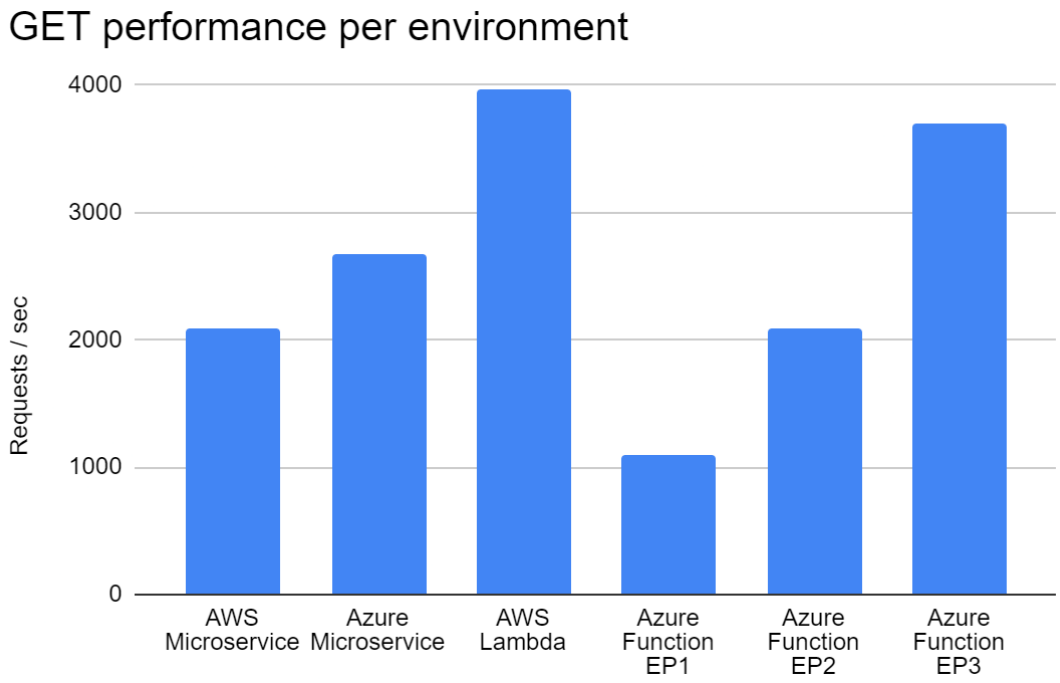

Figure 4.18 GET requests per environment

From Figure 4.18 In case of GET requests it seems that Azure and AWS environments for single microservice instance setups are relatively close. The difference in throughput could be intermittent. However, it looks like private networking performance could be different due to public sharing of the infrastructure. None of the machines were set up on dedicated hosts and therefore could be a subject of bandwidth limitation of private networking. In case of stress testing microservice solutions it looks like Azure yields slightly more performance as opposed to the AWS environment.

It seems that none of the gathered results come close to the performance AWS Lambda provided, except for the most upgraded plan in Azure Function.

4.8.2 Workload A - Database performance

The difference in performance could be investigated further when looked at various latency metrics recorded with each test. As designed each test also yields smaller sets of metrics with recorded time measurement for executing a call to the database server or executing a cache server. Figure 4.19 shows exactly the recorded latency to the database. The latency was recorded at different times and was simply compressed into equal average intervals to be presented on a graph.

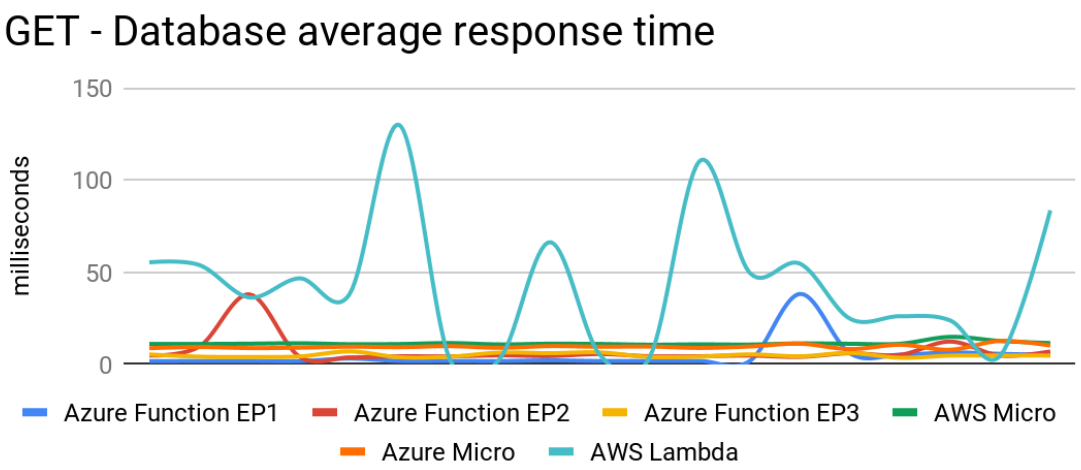

Figure 4.19 GET request - database latency per environment

Database latency could be significant when performance results are very close, but in this case, it looks like between microservice solutions the database latency does not make a difference. Even if accumulated in this case the database latency remains almost the same, with a global average difference of 1 millisecond (Figure 4.20).

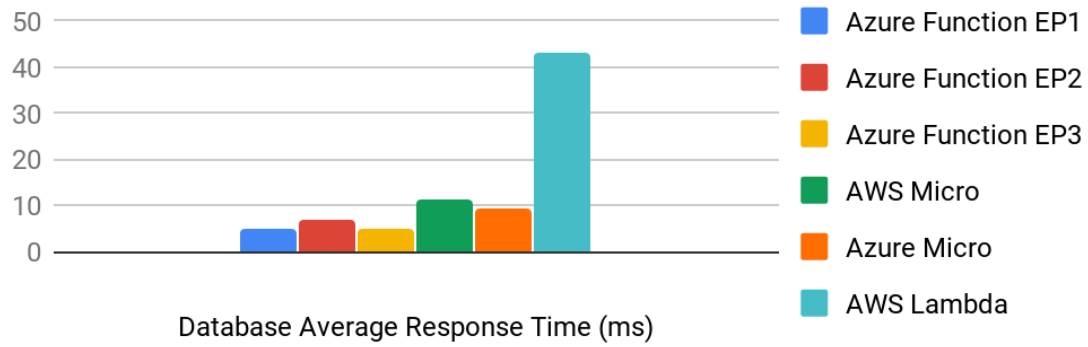

Figure 4.20 GET request - average database latency per environment

What is also interesting when looking at Figure 4.20 is spotting how poorly the AWS Lambda performed when connecting to the database to issue a very simple query statement. Usually, when outliers like that are spotted, they could be excluded not to skew results but in this case the behavior seemed semi-consistent as results began to look better the longer the test was running. More importantly, when looking at this graph and then looking back at the overall performance of AWS Lambda one would think how this could be possible. One explanation that could be drawn from this is the incredibly high availability of the Lambda and the concurrent executions triggered by AWS API Gateway. The first spike of concurrent executions in Figure 4.21 depicts this exact test.

In short, during the test of GET requests the high concurrency of the Lambda outgrew the shortly spiking latency executions. The number of requests executed by AWS Lambda is also a helping factor of this judgement.

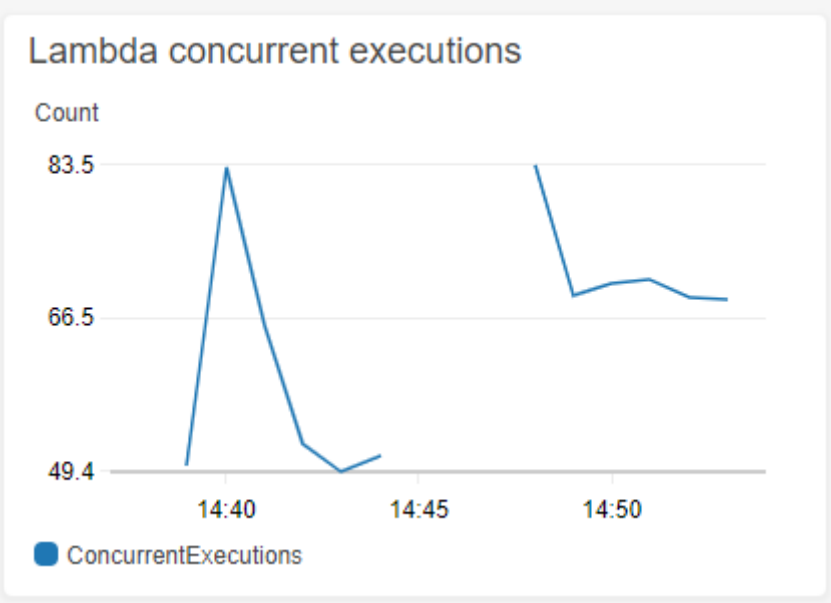

Figure 4.21 Concurrent requests during testing

4.8.3 Workload A - Cache performance

Another important metric to look at is how well the service managed to connect with Redis cache server. As seen on Figure 4.22 the cache response time seems to have little impact on any tested environment. In fact, as opposed to AWS spiking frequently on database connections, connections to Redis cache server improve quickly, ending up being the fastest the longer test has run. A small spike happens in the middle of the test for Azure EP1, it is speculated that the time when the requests started to mainly retrieve data from Redis cache they may have introduced a heavier load on Redis.

In case of difference between Azure and AWS (green & orange, Figure 4.22) microservice solutions, it seems that yet again the difference is way to little. The difference is so little that summing up the latency will not make any significant difference that will indicate as to why this difference occurs.

Overall, it looks like AWS Lambda performance on connecting to Redis is far superior to any other environment. A difference between 1 millisecond and 10 sounds small but a ratio of 10:1 in performance the longer the test is executed, sounds very convincing

and at the very end adds up into a lot of saved time for transferring data. This is also a clear indication as to why Lambda has performed so well, since only a small portion of GET requests at first go over to the database, the rest of the requests after that will only go to the cache and return immediately. This not only saved a lot of time on data transfer but at the very end, has shown its superiority in the amounts of processed requests.

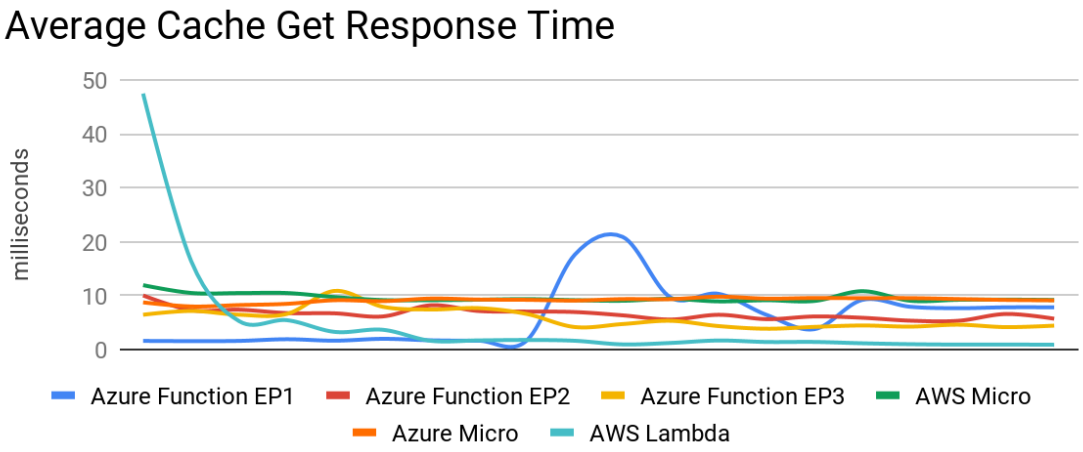

Figure 4.22 GET request - cache retrieve response time

One more metric for cache connectivity during this test is the response time recorded when data which initially was retrieved from the database, after which it had to be saved inside the cache. Repeating again, the purpose of this is to avoid redundant database calls and use cache instead, as per microservice guidelines. In Figure 4.23 it is observed yet again, the initial spike in response time which could be interpreted as a cold-start. Except it only affected the AWS Lambda environment which after a short time resolved itself and the average became closely similar to other environments, without doing it any better. Looking at the performance of this call gives little insight as it only happened for a short while until the data was never saved to cache again since the lifetime of cached data was longer than the test itself.

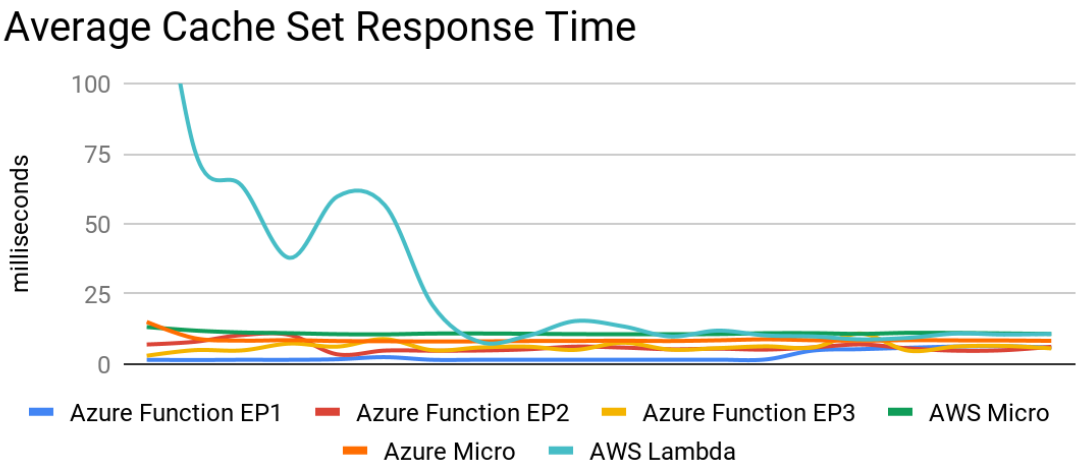

Figure 4.23 GET request - cache saving response time

4.8.4 Workload A - Overall pipeline performance

Last metric worth mentioning is the recorded averages of response time of the whole GET request for every environment. With Figure 4.24 it is possible to confirm the well-known problem of cold-start with AWS Lambda. But FaaS service is not the only one which seems to have this problem. Microservice solutions also, on a small occasion, happen to have smaller performance than usual due to the hosting service remaining idle and cleaning up resources it doesn't currently use, such as registered singleton services and awaiting requests.

At the very end all environments seem to have performed similarly. Yet again AWS Lambda had occurrences of cold start problems and after a little while became more efficient than microservices. The reason why Azure functions did not show any cold start problems is because Azure Functions have in-built a warm-up system where the function service is consistently called but no function is actually executed. This is similar to how microservices are periodically checked by application load balancers to validate whether the endpoint created by the load balancer has the possibility of actually calling the microservice without problems.

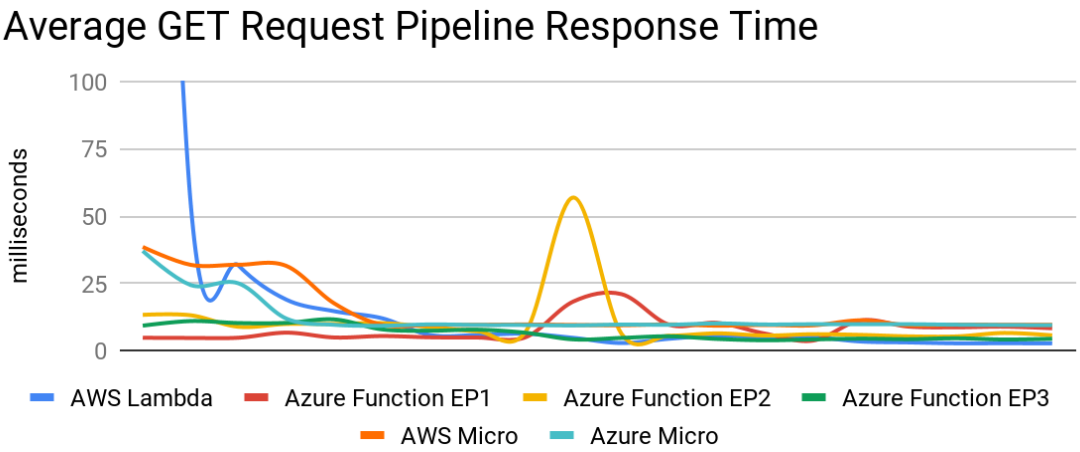

Figure 4.24 GET request - average response time

4.8.5 Workload B - Performance per environment

The exact same benchmarking setup from Table 4.2 was used to test PUT requests.

In case of PUT requests as seen on Figure 4.25 the story is proportionally similar. AWS and Azure environments with microservice setups perform closely with Azure being again, slightly better in terms of overall throughput. This could be a coincidence, but since a similar scenario happened with GET requests it further confirms the higher availability of Azure virtual networks. It could also mean that the accumulated response times during both caching and database calls largely contribute to the overall performance. However, as seen with GET requests this seems like an unlikely case.

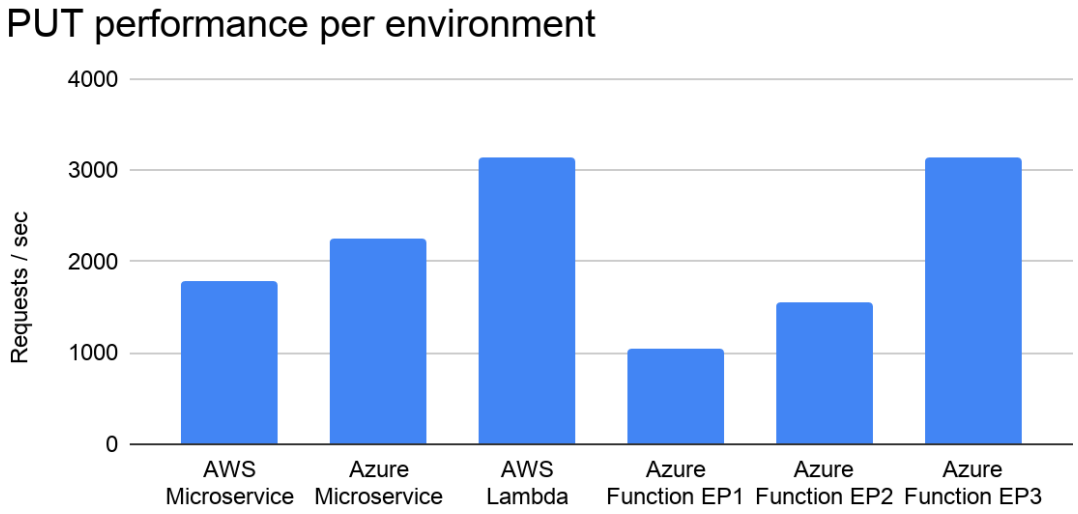

Figure 4.25 PUT requests per environment

On another note, and once again, AWS Lambda is far superior to other environments. But also goes in toe to toe with the best available Azure function plan. From Figure 4.21 it is already known that concurrency has a big impact on how much performance can be gained from executed Lambda functions. The second spike from Figure 4.21 also shows the concurrent invocations caused by a test using PUT requests. The concurrency value is averaged per minute and somewhat matches to how benchmarking was set up in Table 4.2. The number of connections, in this case 100, means that WRK benchmarking tool tries to maintain 100 open sockets which are used to issue requests from. This would also mean that no more than 100 executions could happen at the same time as each thread carefully waits for each request to arrive back and record relevant metrics.

When making the same assumption regarding similar differences in number of requests per second as with GET requests other metrics have to be looked at as well. Figure 4.26 shows a similar trend observed with GET requests. This time the cold-start problem is more visible, with Azure Function performing a little better than the rest.

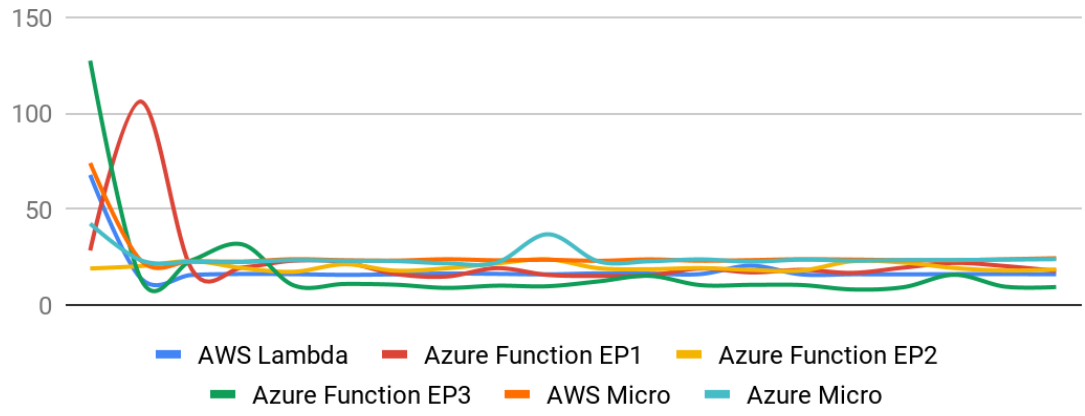

Figure 4.26 PUT request - average response time

The PUT request type is inserting data into existing records inside the database. When looking at the database average response in Figure 4.27 a similar observation can be made to the previous figure.

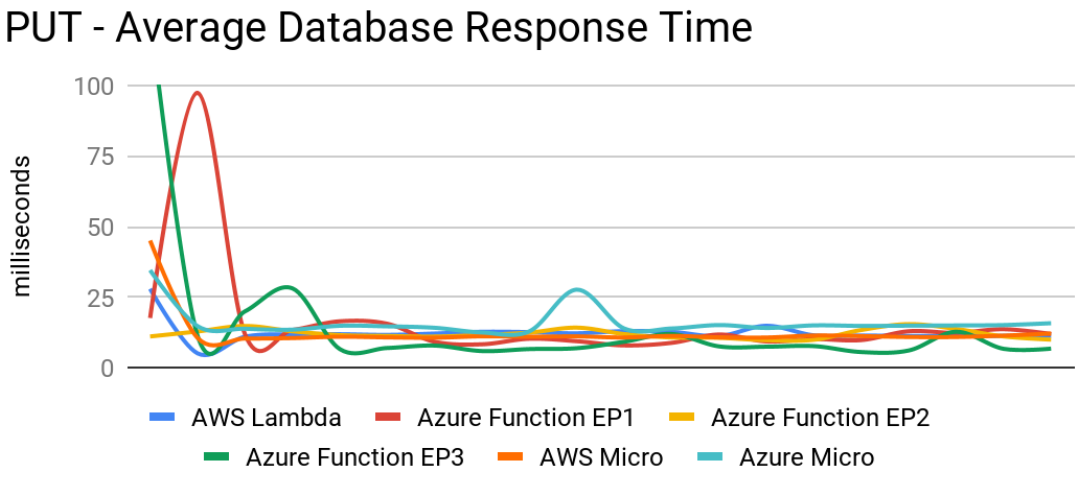

Figure 4.27 PUT request - database latency per environment

Since there is a big similarity in the trend between the two request types, more results can be seen in Appendices of this paper.

4.9 Cost analysis

Now that results have been deeply discussed it is time to find out the cost of running technologies used by the benchmark. To evaluate costs correctly it is important to only calculate costs for technologies that are used for the technologies in question. For example, database related costs are not taken into account as the costs are the same for environments hosted by a given cloud provider. To elaborate further, in the design of a microservice solution and FaaS solution for AWS, the only technologies that change throughout the process are related to the service implemented as either Microservice or FaaS function.

Technology	Type	Cost
AWS VM (Microservice)	t2.micro	$0.0126/hour
Azure VM (Microservice)	B1s	$0.0118/hour
AWS VM (Benchmark)	t2.micro	$0.0126/hour
Azure VM (Benchmark)	B1s	$0.0118/hour
AWS Lambda	N/A	$0.20 per 1M requests $0.0000166667 GB-second
AWS API Gateway	N/A	$3.50 per 1M requests

Azure Function	EP1	$0.25/hour
Azure Function	EP2	$0.50/hour
Azure Function	EP3	$1/hour
AWS Elastic Load Balancer	Application	$0.0252/hour $0.008/LCU-hour
Azure Load Balancer	Standard	$0.025/hour $0.005 per GB

Table 4.3 Technology cost

Table 4.3 lists costs associated with running previously mentioned technologies as well as integrated services that are part of the architecture natively in the cloud. The pricing models differ depending on technology used, e.g. in case of Lambda an online tool for calculating total costs sheds some more light on how exactly prices are calculated.

Environment	Included technologies
Azure (microservice)	VM (per hour) + Standard Load Balancer (per hour, per GB)
Azure (FaaS)	Azure Function (per plan, per hour)
AWS (microservice)	VM (per hour) + Application Load Balancer (per hour, per LCU)
AWS (FaaS)	AWS Lambda (per request) + API Gateway (per request)

Table 4.4 Technology groups

With Table 4.4 it is shown how mentioned groups of technology are calculated together. Again specific things like pricing per virtual network are not taken into account since the purpose of this experiment is to find costs of the two interchangeable architectures. Many technologies have costs broken down as listed in Table 4.4. For AWS Load Balancer the LCU stands for Load Balancer Capacity Unit and is based on usage of the load balancer. It can be based on connections per second, data transfer or rule evaluations - whichever was used the most.

4.9.1 Cost of performance test

Diving deeper into costs it is time to find out the cost of running the actual test. Table 4.5 gives a more detailed view of how much costs have been generated by running test issuing GET requests only. This, by far, was the most performant test in this paper

therefore it is good to take a closer look at how expensive it is to run architectures associated with this test.

Test	Data	Total requests	Cost / test duration (5 mins)	Total $
Azure Micro	218.58 MB	625507	(0.0118 / 12) + (0.025 / 12) + (0.005 / 4)	0.00431
AWS Micro	184.31 MB	804199	(0.0126 / 12) + (0.0252 / 12) + ((4 x 0.008) / 12)	0.00581
Azure EP1	160.78 MB	327531	(0.25 / 12)	0.02083
Azure EP2	281.12 MB	626543	(0.5 / 12)	0.04166
Azure EP3	498.80 MB	1108687	(1 / 12)	0.08333
AWS Lambda	493.05 MB	1191355	0.49 + 4.1697	4.6597

Table 4.5 GET request total costs

For ALB in AWS the LCU value is 4 with 0.008$ per hour as according to the pricing calculator from AWS. Also, it looks like it is based on connections per second rather than data per hour like it is done in Azure. Costs related to AWS Lambda were also done using the calculator provided on the main AWS site.

From Table 4.5 it is important to note that most of the costs except for the Lambda are almost static. Setting up virtual machines and Azure Functions with a premium plan introduces stable costs per each hour of usage with minimal costs related to data transfer for load balancers. In the case of AWS Lambda costs are only generated as the technology is used. This also applies to API Gateway along which Lambda is working with.

AWS Lambda poses a significant difference in pricing out of all pricing models but is easier to calculate and more transparent since the concept usage based on time is taken out of the picture. While flexible and clear on pricing it still seems expensive.

4.10 Cost for performance

With generated total costs per each GET test, the total cost can now be mapped to the average request per second giving an overview of the cost for given performance. For test comparison, the PUT benchmark is also added and shown in Table 4.6.

4.10.1 Based on GET and PUT tests

Test Environment	Req/sec A (GET)	Cost ($) / sec A (GET)	Req/sec B (PUT)	Cost ($) / sec B (PUT)
Azure Micro	2679	0.00001436	1782	0.00001436
AWS Micro	2084	0.00001936	2251	0.00001936
Azure EP1	1091	0.00006943	1036	0.00006943
Azure EP2	2087	0.00013886	1559	0.00013886
Azure EP3	3694	0.00027776	3142	0.00027776
AWS Lambda	3969	0.01553233	3140	0.012296

Table 4.6 Test average performance costs

When calculating benchmarking costs of the two 'heaviest' tests there seems to be little difference since most of the costs across environments are associated with how long the architecture was used for. However, the biggest difference can be observed between AWS Lambda costs. Although the word 'biggest' loses its meaning since the average requests per second is similar. On average there seems to be little difference between tests in terms of costs. However, the performance difference is subjective to the type of test performed and comparing designed tests between each other is not the main goal. What is the main goal however, is comparing the performance and costs generated by microservices and FaaS?

4.10.2 GET cost performance comparison

To further elaborate on performance and costs generated it is worthwhile contemplating on how they transfer in terms of percentages and how environments compare to each other.

Test Environment	% of AWS Lambda performance	% of AWS Lambda cost	% of AWS Lambda cost (Lambda only)	% of Azure EP3 performance	% of Azure EP3 cost
Azure Micro	67.5 %	0.092 %	0.88 %	72.52 %	5.17 %
AWS Micro	52.5 %	0.125 %	1.18 %	56.42 %	6.97 %

Average	60 %	0.1085 %	1.03 %	64.47 %	6.07 %

Table 4.7 GET benchmark performance comparison

Table 4.7 represents differences in performance for GET benchmark. The comparison is done between microservice solutions developed in both AWS and Azure and are compared against AWS Lambda and the highest tier of Azure Functions.

The results suggest that on average AWS Lambda yields around 40% more performance as opposed to single instanced microservices, however it also yields about 99.9% more costs if used the same way as in this test. What is more interesting is that 89.5 % costs of the Lambda solution is caused by the usage of API Gateway (REST). Removing the API Gateway cost as seen on Table 4.7 is creating around 98.9% bigger costs instead of 99.9 %.

On the other hand, Microsoft Azure seems to be a better alternative in terms of costs, creating around 93.9% bigger costs than microservices. It also yields around 35% better performance than microservices.

From generated results it is seen that FaaS technology is more performant than traditional microservices, however they bear much bigger costs.

4.11 Chapter summary

This chapter covered some implementation details deriving from design from Chapter 3. It discussed differences occurred across environments and how certain implementation details also differed between coding solutions. Overall, AWS Lambda and Azure Function solutions had little differences between each other, and the structure of the code remained the same for microservices as well.

The chapter also covered how some of the automation scripts work, especially in terms of running the benchmark and retrieving generated logs. The logs contained important results that could have not been generated in a different way more efficiently.

Retrieved logs helped in generating results across benchmarking tests which lead to many interesting perspectives of how performance changed during benchmark. Based

on generated graphs some speculations were drawn referring many times to a well-known cold-start problem.

Processed results helped in generating cost reports which at the very end helped concluding on which architecture yields more performance and which generates the least amount of costs.

Generated results of this chapter point out two big differences. The throughput of processed events was almost twice as more of that generated by microservices. However, the cost of FaaS was substantially bigger making them less cost-effective. There was not much difference in throughput between AWS Lambda and Azure EP3, the two FaaS solutions. There was a small difference in throughput between microservices solutions for both Azure And AWS environments.

The cost of FaaS could have looked better if more expensive versions were used for microservice virtual machines. However, even with more powerful machines, FaaS is still predicted to generate more costs with this setup. An important note has to be made that AWS Lambda setup uses API Gateway which is an expensive component on its own. This was the only viable option to be used with HTTP workloads as the capabilities of using a Load Balancer with AWS Lambda are limited.

The outcome of performance results suggest that FaaS architectures are not a good fit within a performance environment in terms of costs, or at least not to be used as frequently as in this test.

The next chapter aims to conclude on the research conducted in this paper, putting emphasis on the gathered results and how they reflect on the problem that this paper tried to solve.

5 Conclusion

5.1 Research Overview

In this research a comparison of a service designed using FaaS and microservice paradigms was made. With supporting research from Chapter 2, both paradigms were discussed in the scope of cloud computing and how well do they fit into the SOA model. Investigation from this chapter has concluded on similarities and differences between both paradigms which were helpful in identifying a proper testing practice to answer the research question - benchmarking. To design a proper performance test, a PCP (Pietraszewski's Cost-Performance) benchmarking suite has been proposed and designed in Chapter 3. Some implementation details as well as environment differences, especially those between Microsoft Azure and AWS were also marked and discussed in Chapter 4. Also, in that chapter, with the use of custom designed PowerShell scripts for automation, benchmarking was conducted and results including cost analysis were discussed and analyzed.

5.2 Problem Definition

The aim of this research was to find out if a service designed using FaaS, provides a better performance of processed events and yields fewer computing costs compared to a service designed as a microservice. More precisely, a cost-to-performance analysis was required in order to establish if FaaS is indeed a good replacement of a traditional microservice when designing services in the cloud. Both microservices and FaaS were evaluated using PCP benchmarking, a framework and a set of specially selected workloads tailored to stress HTTP based services.

The outcome of the benchmark discovered that within the cloud, FaaS is indeed a dominant component in terms of throughput of processed events. This performance did come at a cost though. In AWS, FaaS requires various integrations to operate properly, with HTTP workloads an API Gateway is the obvious choice. However, it increases the costs of the architecture, similarly to how only 'premium' plans for Azure Function can compete with the performance of AWS Lambda.

Based on that it is safe to say that services designed using FaaS do yield more throughput of processed events, but they also bear more computing costs.

5.3 Design/Experimentation, Evaluation & Results

The application designed for both microservices and FaaS had differences, with the biggest one being hosting. Microservices were deployed on individual virtual machines with load balancers standing in front. For best performance, AWS Lambda required API Gateway (REST) to be used as a frontline while Azure Function supported HTTP invocations natively without introducing any other services.

The nature of FaaS implementation instructs developers to create function applications without sharing concerns around the hosting capability of the server running FaaS technology. Because of this, in microservices it was decided to use standard kernel hosting in hopes of not introducing any proxy processing. In .Net Core, if looked from a development perspective, specifying a target 'method' as an entry point behaves similarly to how Main() functions work in standard programs. Between Azure Function and AWS Lambda a difference occurs in dependency injection and what objects are passed into or leaving the function.

With the use of WRK benchmarking tool, a set of carefully designed tests were developed and automated with PowerShell scripts in order to best utilize the available cloud resources without introducing unnecessary costs. With WRK being a tool for testing HTTP workloads, the benchmarking focused on the most heavily used request types of GET and PUT. Performance results generated by the tool and additional logs from deployed services were retrieved in an automated way and analyzed locally.

The results from the analysis suggested FaaS technology to yield more performance, with an overall 40% difference in performance for AWS Lambda and 35% for Azure Functions. However, costs associated with running these two FaaS technologies are way bigger than associated costs with running microservices.

The answer to the research question has to be answered partially. With a clear performance lead of the service designed using FaaS, FaaS does indeed yield better performance of processed events. However, with such heavy usage of FaaS, both cloud

providers price the FaaS technology as much more expensive than services designed using microservice architecture. Therefore, the second part of the answer is no, FaaS technology does not yield lower processing costs than those produced by microservice architecture.

To answer this question, many possible ways of benchmarking were considered. Based on the current state of research and trends, benchmarking based on HTTP workloads seemed like the best fit for web services. Combined with microservices and FaaS as targets, this research utilizes some of the best state of the art technologies to ensure the question within this paper is answered well.

The answer to this question could be affected in many ways by changing parts of the underlying design. For example, data that was transferred between systems could be increased in the form of bigger records in the database e.g. encoded pictures. With data taking longer to transfer, the throughput of both microservices and FaaS architecture could drop. In the case of FaaS, this would result in lower costs, bringing it closer to the costs associated with running microservices.

Another way to explore this area would be redesigning the communication method along with benchmarking tools. For example, exploring the same experiment but with AMQP protocol used along with queueing broker technologies such as RabbitMQ.

5.4 Alternate ways of addressing the issue

The experiments conducted in this paper utilize few of the biggest cloud providers widely available today; Azure and AWS. One of the main reasons for choosing them was the budget available at the time of conducting the research. With more available funds and perhaps a wider variety of cloud providers this experiment could be as well designed to work on cloud providers such as IBM or OpenStack. The only real problem associated with that would be the availability of the FaaS platform of these cloud providers, and of course the knowledge required in order to set up the infrastructure.

The designed experiment is also not limited to the chosen data storage and management technologies. For example, the self-managed MSSQL server could be easily swapped over with technologies available by cloud providers such as Amazon RDS or Azure SQL

Server. Open source technologies such as MySQL or MariaDB would also work. However, considering the amount of data that was used throughout the experiment and the type of operations performed, the choice of permanent data storage is not expected to make too much difference.

The choice of caching technology, however, could make a difference. Instead of a self-managed Redis cache server it would be interesting to see the cloud provider solutions such as Azure Cache or AWS Elasticache as well as other open source solutions such as Memcached. However, with these technologies this experiment would notice most of the difference from a data transfer perspective rather than data storage.

5.5 Limitations

When it comes to performing any type of experiment using architecture hosted by cloud providers there are limitations. These limitations mostly appear from the knowledge perspective. Many cloud-based solutions such as FaaS are well documented but only from the usage perspective. To know exactly how such technology works behind the scenes, one would have to either know someone who works for these cloud providers with relevant knowledge or wait patiently until some representative of these companies will appear at a convention or conference, issuing a presentation on the given topic.

Another limitation which affected this work is the budget. As with all cloud providers conducting experiments which are performance heavy are not good for the pocket. The free tiers offered by the two chosen ones were a great help but only initially, during the development stage. At the testing/evaluation stage the free tier budget quickly became not enough and further testing had to be halted.

5.6 Contributions and impact

Main contribution of this work is spiking the interest in FaaS technology and finding the best way of introducing this technology in business scenarios where services are broken down into microservices. Breaking down services into smaller components can be difficult and this research solves the mystery of how interchangeable microservice and FaaS are with each other. To further help in considering, this research is comparing the

two technologies under the context of performance and conducts benchmarking to prove FaaS has comparable efficiency to traditional microservices.

Taking big companies such as Amazon as an example, the concept of microservices is easily transferable into their own product of AWS Lambda. While other companies may not have such luxury, this research also conducts experiments on Azure Functions, drawing differences between the two. Ultimately while this work was unable to prove that FaaS technology is a good price alternative the superiority in performance and ease of development work might pose a good consideration in terms of less frequent workloads. As it stands currently, intentions of FaaS technologies are clearly aimed towards operations that need to happen 'on occasion', especially with AWS Lambda, introducing costs only when technology is actually used.

5.7 Future work & recommendations

Performance environments require careful understanding of underlying architecture and used technologies. This work was affected by unavailable knowledge of commercial FaaS technologies. In the future work it will be worth visiting technologies that are more open-sourced and have plenty of documentation not only on how to use them but about the underlying architecture. Such knowledge could be beneficial when analyzing and figuring the limiting features and inner components of FaaS technology. Work in this paper was also affected by the costs involved with using FaaS technologies. If the cost of using these technologies was not a limiting factor, the experiment could be run many times producing more reliable results.

FaaS technology still has a long way until it is mature enough to provide affordable alternatives in performance oriented systems. Comparing such technology in terms of performance bears many challenges and the judgement included in this paper could be improved further. Some recommendations for future work include:

1. Working with more open-sourced solutions such as Apache OpenWhisk, or Fn Project. These technologies are better documented and can still be utilized in public cloud environments. A huge difference in documentation and inner workings could lead to better understanding of why FaaS technology is more

effective in a performance environment. It could also lead to better understanding of costs which are associated with running a self-hosted FaaS environment.

2. Benchmarking using a wider variety of virtual machines and multiplying their amount. This means choosing from a wider variety of available machines, with increased virtual CPU and memory units. For example, in AWS certain machine tiers like t3.medium have a processing limit that is renewed with each hour. To avoid any performance degradation such machines should be avoided for longer experiments than the one in this paper.
3. Including more distributed systems such as databases. More likely in a business scenario, a distributed database is used for many reasons. Having a microservice or FaaS application accessing distributed databases could yield more reliable results as many business practices could relate to such an experiment. It also forms a different question, whether performance systems involving microservice architectures perform better with distributed databases.
4. Benchmarking using different workloads with a combination of heavier processing operations and more demanding database/caching operations, e.g. transferring encoded pictures.
5. Benchmarking using a different communication protocol such as AMQP, along with today's well documented technologies such as RabbitMQ. Utilizing such technology would involve a completely different benchmarking tool as well as cloud based streaming technologies such as AWS Kinesis.
6. Setting up and comparing open-source FaaS solutions mentioned before, such as Apache OpenWhisk, Fn Project, OpenFaaS and more. These provide well documented examples and explain inner technologies used to compose the FaaS framework.

This paper took interest in two popular architectural patterns of microservices and Function-as-a-Service. It compared both architectures using benchmarking and introduced an interesting way of benchmarking HTTP based services using modern technologies and efficient strategies. At the very end FaaS proved to be a more performant alternative to microservices but has shown that using such architecture is much more expensive in comparison. It has also shown that FaaS is still a relatively fresh area with plenty of more research to be done in the future.

Bibliography

Apache Openwhisk, 2020, Documentation, Source: https://openwhisk.apache.org/documentation.html

Fn Project, 2020, Documentation, Source: https://github.com/fnproject/docs

Amazon Web Services (AWS), 2020, Documentation, Source: https://docs.aws.amazon.com/index.html

Amazon SOA Mandate, 2011, Source: https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/

NGINX, 2020, Source: https://www.nginx.com

WRK, 2020, Source: https://github.com/wg/wrk

Locust, 2020, Source: https://locust.io/

Redis, 2020, Source: https://redis.io/

Abad, C. L., Boza, E. F., & Eyk, E. V. (2018). Package-Aware Scheduling of FaaS Functions. *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering - ICPE '18*, pp. 101-106. doi:10.1145/3185768.3186294

Adzic, G., & Chatley, R. (2017). Serverless computing: Economic and architectural impact. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, pp. 884-889. doi:10.1145/3106237.3117767

Alder, F., Asokan, N., Kurnikov, A., Paverd, A., & Steiner, M. (2019). S-FaaS. *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop - CCSW'19*, pp. 185-199. doi:10.1145/3338466.3358916

Aske, A., & Zhao, X. (2018). Supporting Multi-Provider Serverless Computing on the Edge. *Proceedings of the 47th International Conference on Parallel Processing Companion - ICPP '18*, pp.1-6. doi:10.1145/3229710.3229742

Ast, M., & Gaedke, M. (2017). Self-contained web components through serverless computing. *Proceedings of the 2nd International Workshop on Serverless Computing - WoSC '17*, pp. 28-33. doi:10.1145/3154847.3154849

Brenner, S., & Kapitza, R. (2019). Trust more, serverless. *Proceedings of the 12th ACM International Conference on Systems and Storage - SYSTOR '19*, pp. 33-43. doi:10.1145/3319647.3325825

Cooper, F. B., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R. (2010). Benchmarking Cloud Serving Systems with YCSB. *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010*, pp. 1-12. doi: 10.1145/1807128.1807152

Eyk, E. V., Iosup, A., Abad, C. L., Grohmann, J., & Eismann, S. (2018). A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures. *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering - ICPE '18*, pp. 21-24. doi:10.1145/3185768.3186308

Eyk, E. V., Iosup, A., Seif, S., & Thömmes, M. (2017). The SPEC cloud group's research vision on FaaS and serverless architectures. *Proceedings of the 2nd International Workshop on Serverless Computing - WoSC '17*, pp. 21-24. doi:10.1145/3154847.3154848

Gammelgaard, C. H. (2017). *Microservices in. NET with examples in NancyFX*. Manning Publications.

Glikson, A., Nie, S., & Breitgand, D. (2019). Runbox: Serverless Interactive Computing Platform. *Proceedings of the 12th ACM International Conference on Systems and Storage - SYSTOR '19*, pp. 191. doi:10.1145/3319647.3325852

Großmann, M., Ioannidis, C., & Le, D. T. (2019). Applicability of Serverless Computing in Fog Computing Environments for IoT Scenarios. *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion - UCC '19 Companion*, pp.29-34. doi:10.1145/3368235.3368834

Hafeez, F., Nasirifard, P., & Jacobsen, H. (2018). A Serverless Approach to Publish/Subscribe Systems. *Proceedings of the 19th International Middleware Conference on - Middleware '18*, pp. 9-10. doi:10.1145/3284014.3284019

Hall, A., & Ramachandran, U. (2019). An execution model for serverless functions at the edge. *Proceedings of the International Conference on Internet of Things Design and Implementation - IoTDI '19*, pp. 225-236. doi:10.1145/3302505.3310084

Heinrich, R., Hoorn, A. V., Knoche, H., Li, F., Lwakatare, L. E., Pahl, C., Shulte, S., Wettinger, J. (2017). Performance Engineering for Microservices. *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion - ICPE '17 Companion*, pp. 223-226. doi:10.1145/3053600.3053653

Kanso, A., & Youssef, A. (2017). Serverless. *Proceedings of the 2nd International Workshop on Serverless Computing - WoSC '17*, pp. 6-10. doi:10.1145/3154847.3154854

Kaplunovich, A. (2019). ToLambda--Automatic Path to Serverless Architectures. *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pp.1-8. doi:10.1109/iwor.2019.00008

Karhula, P., Janak, J., & Schulzrinne, H. (2019). Checkpointing and Migration of IoT Edge Functions. *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking - EdgeSys '19*, pp. 60-65. doi:10.1145/3301418.3313947

Kim, J., Jun, T. J., Kang, D., Kim, D., & Kim, D. (2018). GPU Enabled Serverless Computing Framework. *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pp. 533-540. doi:10.1109/pdp2018.2018.00090

Kuntsevich, A., Nasirifard, P., & Jacobsen, H. (2018). A Distributed Analysis and Benchmarking Framework for Apache OpenWhisk Serverless Platform. *Proceedings of the 19th International Middleware Conference on - Middleware '18*, pp. 3-4. doi:10.1145/3284014.3284016

Li, J., Kulkarni, S. G., Ramakrishnan, K. K., & Li, D. (2019). Understanding Open Source Serverless Platforms. *Proceedings of the 5th International Workshop on Serverless Computing - WOSC '19*, pp. 37-42. doi:10.1145/3366623.3368139

Lin, W., Bakir, F., Krintz, C., Wolski, R., & Mock, M. (2019). Data Repair for Distributed, Event-based IoT Applications. *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems - DEBS '19*, pp. 139-150. doi:10.1145/3328905.3329511

Lloyd, W., Vu, M., Zhang, B., David, O., & Leavesley, G. (2018). Improving Application Migration to Serverless Computing Platforms: Latency Mitigation with Keep-Alive Workloads. *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pp. 195-200. doi:10.1109/ucc-companion.2018.00056

Nadgowda, S., Bila, N., & Fisci, C. (2017). The less server architecture for cloud functions. *Proceedings of the 2nd International Workshop on Serverless Computing - WoSC '17*, pp. 22-27. doi:10.1145/3154847.3154850

Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., & Josuttis, N. (2017). Microservices in Practice, Part 1: Reality Check and Service Design. *IEEE Software, 34*, pp. 91-98. doi:10.1109/ms.2017.24

Sewak, M., & Singh, S. (2018). Winning in the Era of Serverless Computing and Function as a Service. *2018 3rd International Conference for Convergence in Technology (I2CT)*, pp. 1-5. doi:10.1109/i2ct.2018.8529465

Shahrad, M., Balkind, J., & Wentzlaff, D. (2019). Architectural Implications of Function-as-a-Service Computing. *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1063-1075. doi:10.1145/3352460.3358296

Shao, J., Zhang, X., & Cao, Z. (2018). Research on Context-based Instances Selection of Microservice. *Proceedings of the 2nd International Conference on Computer Science and Application Engineering - CSAE '18*, pp. 1-5. doi:10.1145/3207677.3277954

Vokolos, F. I., & Weyuker, E. J. (1998). Performance testing of software systems. *Proceedings of the First International Workshop on Software and Performance - WOSP '98*, pp. 80–87. doi:10.1145/287318.287337

Yan, M., Castro, P., Cheng, P., & Ishakian, V. (2016). Building a Chatbot with Serverless Computing. *Proceedings of the 1st International Workshop on Mashups of Things and APIs - MOTA '16*, pp. 1-4. doi:10.1145/3007203.3007217

Yussupov, V., Breitenbücher, U., Leymann, F., & Müller, C. (2019). Facing the Unplanned Migration of Serverless Applications. *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, pp. 273-283. doi:10.1145/3344341.3368813

Yussupov, V., Breitenbücher, U., Leymann, F., & Wurster, M. (2019). A Systematic Mapping Study on Engineering Function-as-a-Service Platforms and Tools. *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, pp. 229-240. doi:10.1145/3344341.3368803

Zhang, M., Zhu, Y., Zhang, C., & Liu, J. (2019). Video processing with serverless computing. *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video - NOSSDAV '19*, pp. 61-66. doi:10.1145/3304112.3325608

Zhao, S., Dziurzanski, P., Przewozniczek, M., Komarnicki, M., & Indrusiak, L. S. (2019). Cloud-based dynamic distributed optimization of integrated process planning and scheduling in smart factories. *Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO '19*, pp. 1381-1389. doi:10.1145/3321707.3321826

Appendix A

Workload C results

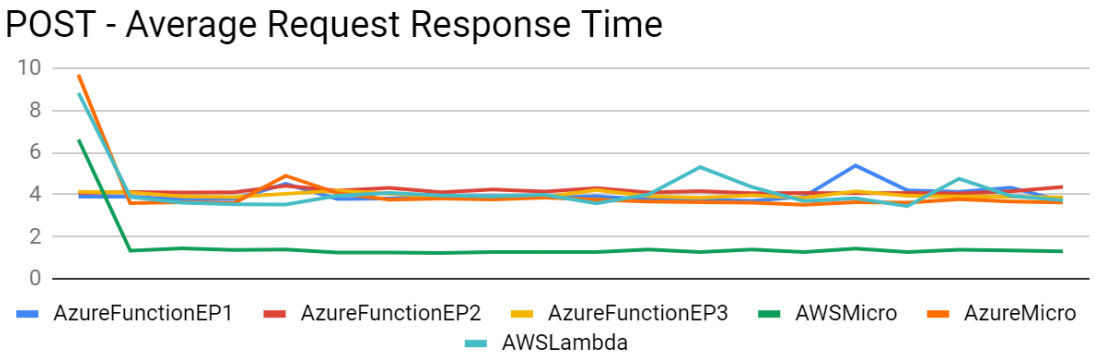

Figure A.1 Workload C average response time

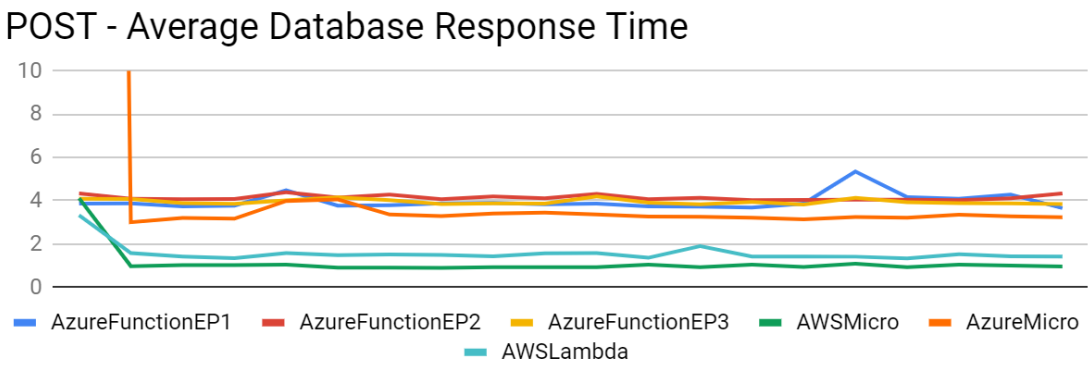

Figure A.2 Workload C average database response time

Workload D results

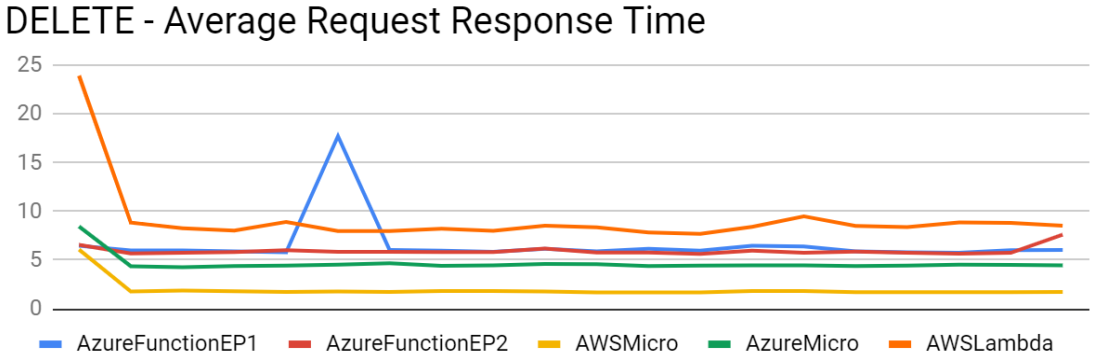

Figure A.3 Workload D average response time

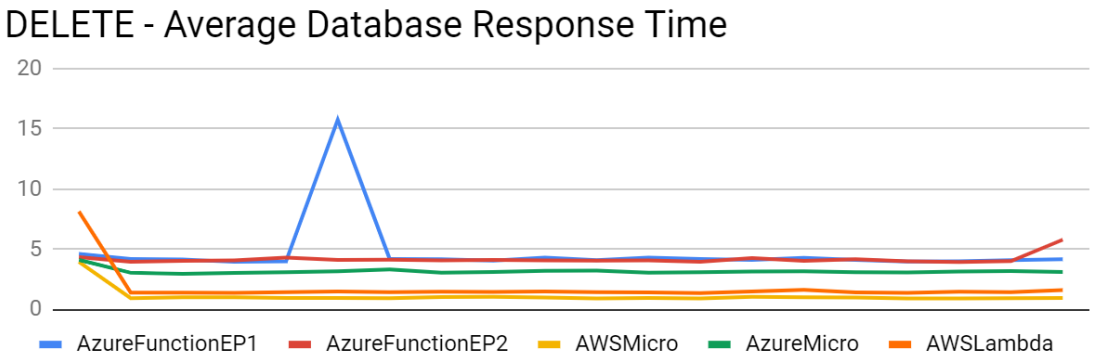

Figure A.4 Workload D average database response time

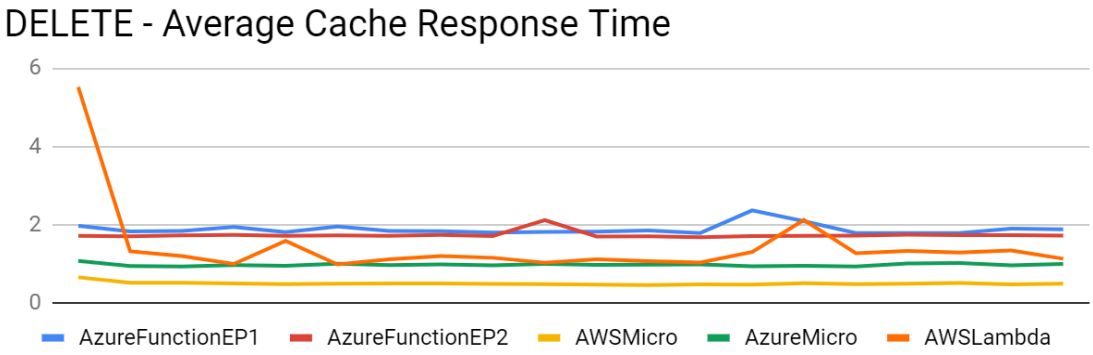

Figure A.5 Workload D average cache response time