UPPSALA
UNIVERSITET

UPTEC F 18006
Examensarbete 30 hp
Mars 2018

Using Function as a Service for Dynamic Application Scaling in the Cloud

Andreas Abrahamsson

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Using Function as a Service for Dynamic Application Scaling in the Cloud

Andreas Abrahamsson

Function as a Service is a new addition to cloud services that allow a user to execute code in form of a function, in the cloud. All underlying complexity is handled by the cloud provider and the user only pay per use. Cloud services have been growing significantly over the past years and many companies want to take advantages of the benefits of the cloud. The cloud services deliver computing resources as a service over a network connection, often by the Internet. To use the benefit of the cloud, one can not just move an application to the cloud and think that it will solve itself. First of all, an application needs to be optimized to be able to take advantages of the cloud. Therefore, together with Tieto, a microservice architecture have been the main architectural pattern when Function as a Service has been evaluated. A major problem with applications, both application built with a monolithic and microservice architecture, is to handle great amounts of information flows. An application may have scaling issues when an information flow becomes too large.
A person using Function as a Service does not have to buy, rent or maintain their own servers. However, Function as a Service has a certain memory and runtime restrictions, so an entire application cannot be applied to a Function as a Service. This thesis examines the possibility of using Function as a Service in different architectural environments and estimating the cost of it. Function as a Service is a new addition to cloud services, so cloud providers are also compared and evaluated in terms of the Function as a Service functionality. Function as a Service has been tested directly on various cloud platforms and even developed and executed locally, encapsulated in containers. The results show that Function as a Service is a good complement to an application architecture. The results also show that Function as a Service is highly flexible and cost-effective, and it is advantageous compared to physical servers and Virtual Machines. Depending on how a function is built, the developer can lower the cost even more by choosing the cloud supplier that fits best for their use. With the flexibility of Function as a Service, applications can handle greater information flow without bottlenecks in the infrastructure and therefore, becomes more efficient and cost-effective.

Handledare: Torbjörn Lundmark
Ämnesgranskare: Andreas Hellander
Examinator: Tomas Nyberg
ISSN: 1401-5757, UPTEC F18 006

Populärvetenskaplig sammanfattning

Ett stort problem med dagens applikationer är hanteringen av det stora informationsflödet. Oavsett arkitektur så kan det skapas en flaskhals i infrastrukturen vilket i sin tur skapar ett skalningsproblem. Ett sätt att eventuellt kunna lösa detta problem är att använda sig utav det nya tillägget till molntjänsten, ”Function as a Service”. ”Function as a Service” ger utvecklaren möjligheten att skriva och köra sin kod i form av en funktion där all underliggande komplexitet tas hand om av molnleverantören. Vilket betyder att dels funktioner i en applikation istället kan köras i en ”Function as a Service” för att göra applikationen dels snabbare men även för att spara minne på den server man använder. Genom att använda ”Function as a Service”, så behöver inte en utvecklare bry sig om att köpa, hyra eller underhålla sina egna servrar. ”Function as a Service” har utvärderats i form av funktionalitet men även skillnaden mellan molnleverantörerna som erbjuder tjänsten. I denna masteruppsats har ”Function as a Service” applicerats till olika arkitekturer, monolitisk- och mikroservice arkitektur, för att framförallt utvärdera möjligheten att skala en applikation med hjälp av molntjänster. ”Function as a Service” har testats direkt på molntjänsternas plattformar och även skapats lokalt. ”Function as a Service” har även utvecklats och körts i containers, för att öka flexibiliteten ännu mer.

Resultaten visar att ”Function as a Service” är ett väldigt bra tillägg till en applikations arkitektur. På grund av dess flexibilitet och låga kostnader finns det många olika sätt och fördelar att använda det på. Jämfört med att köra samma typer av funktioner direkt på fysiska servrar eller på virtuella maskiner så är ”Function as a Service” mer kostnadseffektiv, men framförallt smidigare då man vill ändra storleken på funktionerna. Det finns begränsningar för ”Function as a Service”, vilket betyder att en funktion har minnes- och tidsbegränsningar. Även om ”Function as a Service” är billigt och väldigt lättanvändbart så måste man utvärdera vad för typ av funktioner som ska köras på vilken plattform, för att kunna utnyttja tjänsten fullt ut. På grund av de olika begränsningarna som molntjänsterna har på ”Function as a Service” så kan det skilja sig väldigt mycket i pris. Så det är viktigt att välja den molntjänst som är bäst anpassad för sin funktion.

Alltså, tack vare ”Function as a Service” flexibilitet, så är det ett väldigt bra tillägg till en applikations arkitektur. Det ger utvecklaren möjlighet att skala där det finns flaskhalsar i infrastrukturen. Tjänsten kan både läggas till vid behov till befintliga applikationer men även när nya applikationer skapas och man vill att det ska vara en del av applikationen. Genom att använda ”Function as a Service” kan en applikation bli mer robust, effektivare och även mer kostnadseffektivt.

Contents

1 Introduction

This section focuses on explaining the background to the problem and the objective of the thesis.

1.1 Background

Monolithic architecture has been the general approach for building and running an application. It is a reliable and robust architecture where the application is built as a single unit, [19]. Many companies want to move their applications to the cloud to be able to use all the benefits of the cloud. By using the cloud, the developer does not have to handle any own local hardware, but instead shares from a pool of physical and virtual resources. Thus, the cloud can shortly be explained as computing resources that are delivered as a service over a network connection, [16]. A developer can choose how much the cloud services should take care of and how much they want to handle. Depending on how much the cloud services should take care of, there are different models a developer can choose from. In Figure 1, I show a few examples to provide insight into which models there are to choose from.

SaaS, PaaS, IaaS and On-Premises

SaaS	PaaS	IaaS	On-Premises
Software as a Service	Platform as a Service	Infrastructure as a Service	
Application	**Application**	**Application**	**Application**
Data	**Data**	**Data**	**Data**
Runtime	Runtime	**Runtime**	**Runtime**
Middleware	Middleware	**Middleware**	**Middleware**
Operating System	Operating System	**Operating System**	**Operating System**
Virtualisation	Virtualisation	Virtualisation	**Virtualisation**
Hardware	Hardware	Hardware	**Hardware**

Figure 1: The table shows an example of different models and what can be outsourced to the cloud (the black text) and how much you need to handle yourself (The blue text).

By running the application on-premises, the company must handle everything from hardware to the application itself. Cloud services allow the developer to decide how much they want to manage themselves and choose the model that fits best for use. Figure 1 shows three different models that cloud services offer. SaaS is when a user wants to run an application in the cloud and everything from hardware and application is taken care of by the cloud service. If a user wants to manage more, they can use PaaS. PaaS is when a user controls the application and configuration around the application and allows cloud services to take care of the rest. And then there is IaaS, where a user can outsource hardware and cloud infrastructure but manages the runtime environment, operating system, etc. Which more or less makes it possible to control the entire application but does not need to maintain its own physical servers, [16].

To use the benefits of the cloud service an application architecture needs to be optimized. Basically, the application cannot just be moved to the cloud and solve everything itself. The application must me optimized to be able to run in the cloud, [13]. Therefore, with Tieto [26], I have included microservice architecture in this thesis.

Microservice architecture is not a new invention, many see it as a part of an SOA, Service Oriented Architecture, [13]. Both microservice architecture and SOA use a complex architecture pattern and distributed system and both need to implement a communication between the services, [3]. Microservices communicates with API and in SOA the services communicate with an Enterprise Service Bus, ESB. SOA consists of two main parts of the architecture, service providers and service consumers. The service provider includes all the services that are defined within the SOA and the service consumer includes the human user, third parties, etc. as can be seen in Figure 2.

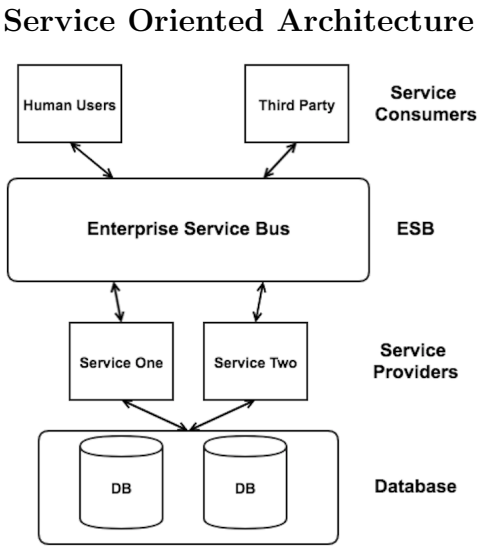


Figure 2: Shows an example of a Service Oriented Architecture

In both microservice architecture and SOA, each service has a specific responsibility, unlike for monolithic architecture, where the entire application is built as a single unit. A drawback with SOA is that all communication goes through the ESB. For example, if one of the services slows down, the entire application can be affected because the ESB is clogged with all requests for just that specific service and in the worst case scenario could cause system failures. As mentioned, microservice architecture and SOA are quite similar but the largest difference is when it comes to size and scope. Figure 3 shows a simple and clear image which shows that a microservice is much less than SOA and an independent deployable service, [17].

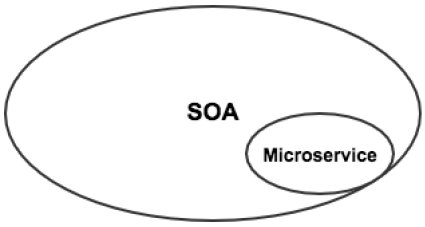


Figure 3: Shows an illustration of how microservice is seen as part of SOA, i.e. as less independent deployable services

A microservice architecture uses suites of small services, and together they create an application. The idea is to make the services even more independent by pushing them into containers and making updates by replacing a service instead of an entire system. By packing a microservice with all its dependence, such as runtime environment, library, code etc. in containers, the microservice is able to be executed anywhere. Containers make it much easier for a user to develop and test a service locally and without any changes use the same container in production, [20].

There are still some difficulties that need to be addressed, one of them is scaling. Even if microservices are smaller services that are independent of each other, they still need a certain amount of CPU, memory, etc, [9] That may be an issue when the application receives a larger information flow. Although it is possible to create a new container with a microservice to handle a particular event, the hardware may still be limited. The idea is to make an application even more cost-effective and time-efficient by adding the new cloud services tool Function as a Service to the architecture.

By using Function as a Service in an application architecture, certain features can be run in the public cloud instead of taking resources from physical servers or virtual machines. This means that the service is hosted by a cloud provider off-site and accessible through a public network, such as the Internet. Function as a Service can be seen as a tool for an application where some of the underlying complexity can be outsourced to the cloud provider, [23]. For example, features in an application that only runs for a few hours a day or just running certain times of the day could instead run like a Function as a Service. This would save space on the physical servers or virtual machines, which could help to scale an application. Alternatively, at a high load of the application, use Function as a Service to handle a part of the scaling.

Is it possible to use Function as a Service as part of an application to make an application more reliable when it comes to scaling and cost-effectiveness? An important aspect to consider is that the function as a service is a relativity addition to cloud services and some questions must be answered before it can be applied in an application architecture.

1.2 Objective

The objective of this master thesis is to evaluate how the concept Function as a Service should be used when developing and deploying a new application based on a microservice architecture. What requirements we should have on Function as a Service in terms of performance, cost, and security. Because Function as a Service is a new addition to cloud computing services, evaluation of current Function as a Service provider includes in the thesis.

1.2.1 Goals

The goals of this master thesis are listed below. The main objective has been to evaluate whether Function as Service can be included in various architectural environments, primarily for a microservice architecture and how to efficiently use Function as Service benefits.

- Define an architectural pattern on how Function as a Service should be added to a microservice architecture.
- Define and build a prototype for evaluation of architecture pattern.
- Investigate the concepts, characteristics, and performance of Function as a Service.
- Perform a study on how Function as a Service shall be monitored and orchestrated in different environments and applications.
- Additional: investigate the impact of developing with Function as a Service seen from an economic perspective, how to manage resource utilization early in the design lifecycle.

2 Theory

This section is dedicated to the theory behind the thesis. Understand different application architecture and explain the main focus of the thesis, Function as a service. There is also a brief introduction to containers, as well as cloud suppliers and their price plans.

2.1 Microservices

"The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process", [9].

A few years ago a new terminology with the name microservices was introduced. It is not a revolutionary topic because some say it is part of already-existing SOA. The idea of microservices is to decentralize an application, basically dividing an application into smaller services that can be deployed and maintained independently, [13]. The microservices are scaled by distributing these services across servers and can be replicated as needed, [9].

A Monolithic- and a Microservice architecture

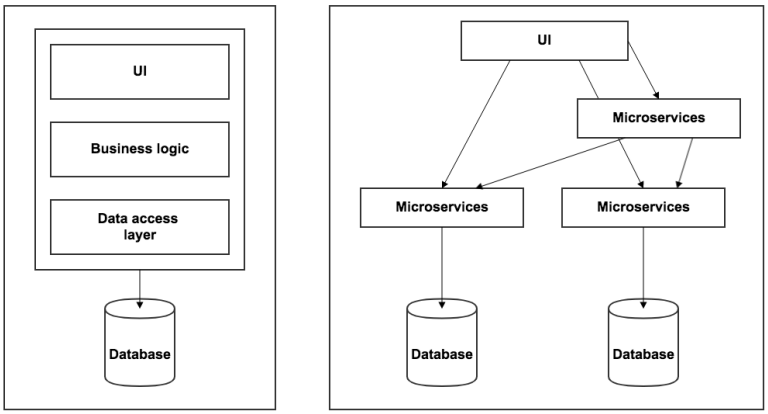


Figure 4: Shows an example of how a monolithic- and a microservice architecture can be built

A monolithic architecture is a classic and the most common way to build an application and it is structured as a single unit, Figure 4. Microservices should not be viewed as a replacement for monolithic applications, as monolithic architecture still is easier to build and distribute for smaller projects. Distributed applications like an application based on microservice architec-

ture are more complex to build, [3]. Each microservice has its own responsibility and communicate with each other via a REST API, [19], which is explained later in the section. The partial deployment makes microservices unique because each microservice can be maintained and upgraded independently, and new ones can be constructed when needed. If we compare a microservice architecture with a monolithic architecture, we can see that monolithic applications consist of three main parts, Figure 4. There is a user interface (UI), a server-side application (Business logic, data access layer, etc.) and a database, [18]. When a monolithic application needs updating, a completely new version must be deployed. With a microservices architecture where each functionality is placed in a separate service, an update can be done by updating only the specific service without affecting the entire application. Therefore, a microservice architecture is much less likely to cause system failures, [22].

A microservice can be written in all programming languages and can use different data storage technologies. A microservice can be directly linked to its own database or different microservices can share a database and use each other stored data, [19].

As the name microservice explains, there are micro (small) services that together build an application. But how small should a microservice be? Depending on who who gets the question, there will be different answers, as a microservice does not have a certain size. The general idea is that each service should be managed individually by a group with a specific task, where each group should not contain more than a couple of people, [9].

A microservice can be deployed directly on a physical server, virtual machine (VM) or in a container. The benefits of using microservice in a container are that it encapsulates the microservice and that it makes it more independent, [15]. The container makes it easy for the user to package, design and execute the code anywhere, which is advantageous when developing and deploying microservices, [2]. You can create your container containing your microservice and test it locally on your own computer, and without making any changes, you can distribute the same container to your application. The communication between microservices is made by an API (Application Programming Interface) and the developer chooses what the availability of the various microservices should have. An API is used to read and write data between application components, [21]. An API can be viewed as an interface that has a set of features that allow users to access specific data in an application. The Representative State Transfer (REST) API is a way to communicate between application components via the web. REST API is based on Http-requests and the Web API is a great way to use it in a .Net-framework, [19].

2.2 Function as a Service

Function as a service, or FaaS, is a relatively new additional service to existing cloud services. It can be seen as a tool that can be used both for existing and new applications. The idea is that a user can develop, operate and maintain its own code without having to handle the underlying complexity, [23]. The cloud services provide a user with all the infrastructure, from servers to scaling. All that a user needs to do is create a function triggered by a particular event and the rest takes care of the cloud provider, [1].

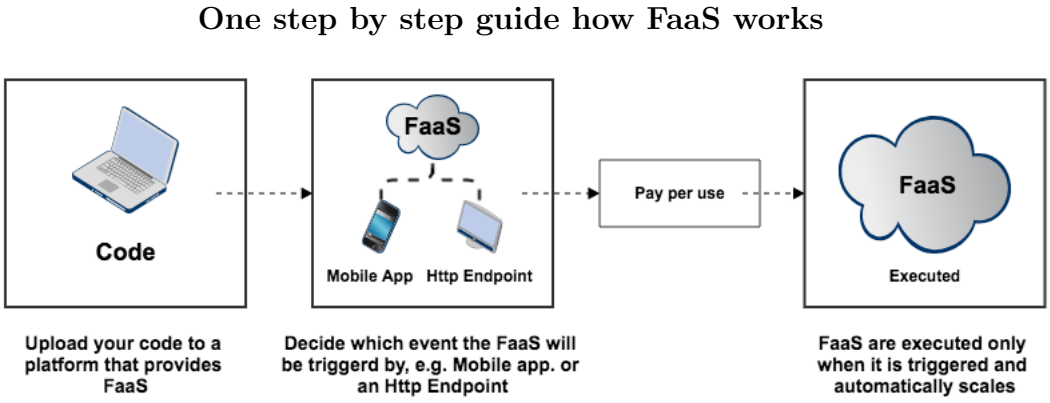


Figure 5

As figure 5 demonstrates, a user uploads a code in the form of a function to a cloud service that provides FaaS or writes the function directly to the cloud service platform. An event is determined to trigger the function when needed. When a function is triggered, it is executed within a few milliseconds and then shut down immediately after it is ready. What makes FaaS advantageous to compare with other options is its cost-effectiveness. All that a user is charged for is when a FaaS is running, so when a function is not used, the user is not charged. Since all underlying complexity is handled by a cloud provider, users have the choice to choose from many different programming languages to develop their function in, [14]. FaaS is an event-driven service with many different events to choose from. For example, an event might be a timer trigger if you want a function to trigger at a specific time or a queue system that starts a function when the queue becomes too long. These examples are ideas on how to move functions to the cloud as well as handle scaling problems. A function can also be triggered by an Http-request, which is the most common event in this thesis, [23].

However, there are some disadvantages of FaaS. As many other cloud services, FaaS also has limits on how long each invocation can run and how

much memory each function may use. The more memory and time a function requires, the higher the cost. If a function is too large or takes a long time to complete, the cloud provider shuts down the function without returning a response, [23].

There are different constraints depending on the cloud providers used. Generally, the time limit is five minutes but the memory limit varies more, $100MB - 3008MB$. It is also important not to do the functions too small because there is also a lower limit on memory and runtime that platform providers charge for. Thus, it is important to develop a function so that is optimized to take full advantage of FaaS, [27].

FaaS uses a function.json file that defines which configurations and bindings a function have. The runtime uses the function.json file to determine which events will trigger the function and the type of data that allows transfer to and from an executed function. In Figure 6 is shown an example.json file example, [1].

Function.json

```
{
  "bindings": [
    {
      "type": "httpTrigger",
      "methods": [
        "get"
      ],
      "authLevel": "function",
      "direction": "in",
      "name": "req"
    },
    {
      "name": "$return",
      "type": "http",
      "direction": "out"
    }
  ],
  "disabled": false,
  "scriptFile": "../AzureFunc.dll",
  "entryPoint": "AzureFunc.Functions.Add"
}
```

Figure 6: An example of a Function.json file that executes a FaaS with an Http-request. Function.json defines the configurations and bindings that a FaaS has

JSON is a good way to store and exchange data, it is only in text format and therefore easy to use and understand. A common use of JSON is when web servers exchange data. JSON can be used when communicating with servers and is also used as a data format for any programming language, [8].

2.3 Container vs VM

If containers are compared to VMs, there are many similarities to how they isolate and allocate. However, a major difference is the use of the operative system. A VM requires an operating system on every VM, and a container visualizes the operating system instead. This means that in every VM that is created, there is a complete copy of the operating system, libraries etc. that tends to slow down a process. VMs use a Hypervisor, Virtual Machine Monitor, which is a program that allows one to host many VMs on a single hardware, [15].

Containers work with layers, where all its dependencies, lite, code, libraries, runtime, etc. are packaged together. Containers that share servers use the same operating system but are still isolated from each other, [24]. Containers are built to take as little memory as possible, so for example if a VM takes approximate GB in memory, a container with the same functionality takes approximate MB instead. However, it is possible to combine VMs and containers which gives the developer many different flexible ways to create applications, [6].

2.4 Docker Container

Docker has been used in this project to build and execute the microservices in containers, including FaaS. Docker is a container platform with an open source technology where you can build and manage your application. With Docker, you can do anything from developing an application to using it in production. It can be applied both on-premises and/or in the cloud. Docker is available to everyone and easy to use, [6].

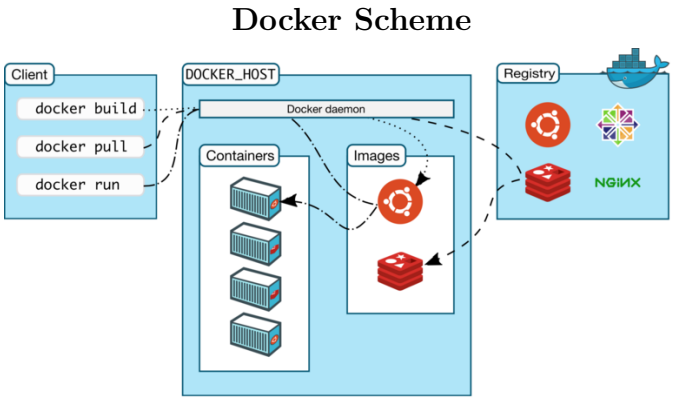


Figure 7: An example of the main parts needed to create a Docker container, [4]

Figure 7 shows the basic components for building a docking container. Docker can communicate with the local computer via the terminal, where the user can build an image or pull an image from the Dockers registry. A container is created when an image is executed, the container's dependencies are defined in the image. One major advantage of a container is that it allows the user to pack an application with all its dependence in a standardized software development device, [24].

Thus, a container is the runnable instance of an image. Users can create and operate a container on their local computer and regardless of the environment, the containerized software will always be the same. The idea of using containers is to isolate e.g. a microservice, to make them independent of the rest of the application. A container could run on both physical servers and virtual machines, or both, and you can also combine containers with VMs if needed, [15].

2.4.1 Dockerfile

An image is a template with instructions for creating a container. It is lightweight and contains everything that is required to run a piece of software. The executable package contains everything from code to system library. An image can be based on another already existing image but with customizations. To build an image, a Dockerfile is created with a set of layers that defines your applications environment. Because the application environment is defined in the Dockerfile, it is possible to reproduce it anywhere. When you change something in a Dockerfile, only the newly added part needs to be updated and not the entire Dockerfile, [6]

An example of a Dockerfile

```
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
ADD . /app

# Install any needed packages specified in requirements.txt
RUN pip install -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]

```

Figure 8: A Dockerfile with different of layers used to create a container, [5]

Figure 8 shows an example of a Dockerfile. It is a description of how to package a service with its dependencies in a container. At the top of the Dockerfile, is the command "FROM", which indicate which runtime environment the container will use. The larger an application is the better it is to keep track of all containers, therefore it can be good to have "WORKDIR". "WORKDIR" allows the developer to define which working directory they are going to work from. Then there is "ADD", where all information needed in the new working directory is added. The "RUN" command installs and runs the packages that are specified, which in this case are the requirement textfile. "EXPOSE" makes the port used available to the world outside the container. At last, "CMD", the command executes app.py in this case when the container launches with python as the communication language. This provides a brief introduction how a Dockerfile can look like, there exist many other commands that suit the needs of a developer, [5].

2.4.2 Docker-compose file

To make it easier to build and run multiple containers, a docker-compose file can be created. It is a YAML file and is a tool for defining and running a multi-container application. With a single command, a user can create and launch all services encapsulated in the containers from the user configurations. For development, the compose file is very good for testing the application environment.

An example of a docker-compose file

```
1  version: '3'
2
3  services:
4    product-service:
5      build:
6        context: ./app
7        dockerfile: Dockerfile
8      ports:
9        - 8080:80
10
```

Figure 9: A docker-composing file that execute a service based on a Dockerfile

Figure 9 illustrates an example of a docker-compose file that executes a service based on a Dockerfile. It is a basic docker-compose file that defines the version at the top. Version 3 is the newest and currently the most beneficial version to be used. Line 3 defines which service to run, in this case, a service called product service. The service is based on an image, where the different layers are set in the Dockerfile. *./App* explain to the docker-compose the file that the Dockerfile is located in this location. In the *./App* direction, the

name of the dockerfile is set to Dockerfile, but the dockerfile can be named anything. The port defines which container port you can access. In this example, the microservice is created and executed in a container with access port 8080.

Figure 10 shows when the docker-compose file is built with the *docker − compose build* command and executed with *docker − compose up −d* command, *−d* is added so that the container will run in the background. As can be seen in Figure 10 all layers from the dockerfile are built and with *docker − compose up*, the container is created with access port 8080, as defined in the docker-compose file.

An example of how to build and run the docker-compose file in the terminal

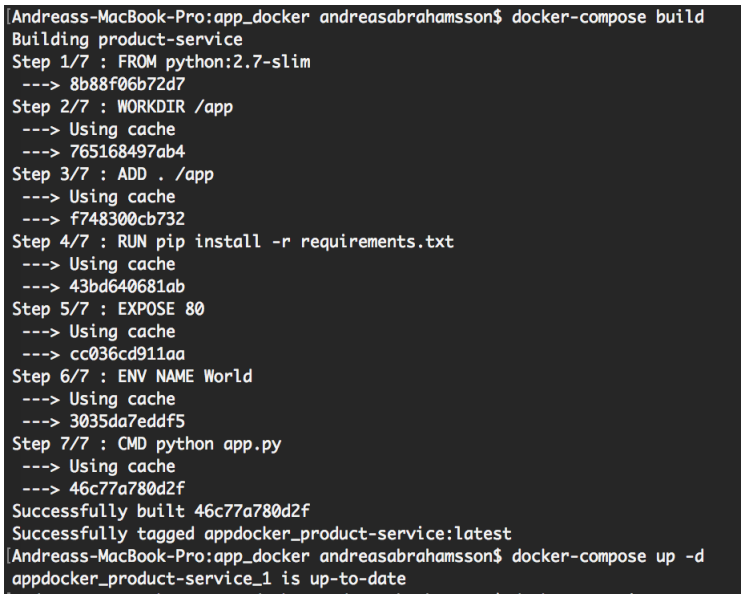


Figure 10: Shows how an image is built and executed to create a container with all its dependencies

2.5 Cloud Services

Cloud services provide an opportunity to build and implement an application in a modern, efficient and reliable manner. Cloud services provide an opportunity to use computing resources that are delivered over a network connection. The company providing cloud services allows the user to decide how much the platform provider wants to take care of, [16]. In this project, three different cloud platforms have been evaluated for their FaaS functionality.

Cloud services providing FaaS

(a) Microsoft Azure Functions, [1]

(b) Amazon Web Services Lambda, [14]

(c) Google Cloud Functions, [10]

Figure 11: The three largest cloud services providing FaaS, their FaaS functionality have been evaluated with respect to their limitations and costs

The three biggest cloud services that provide FaaS are Microsoft Azure, Amazon Web Services, and Google Cloud, [24]. As previously mentioned, no matter what cloud service is used, users should be able to develop their code in any programming language. FaaS is a new addition to cloud services and there are some differences depending on the cloud platform used, which is to be evaluated in the thesis. The differences are in price, memory and execution time. Note, because FaaS is relatively new, it is developing fast and new updates are constantly updated and enhanced.

The first cloud supplier presented is Microsoft Azure, although Amazon was the first major company to develop FaaS, I have almost exclusively used Azure Functions in this project, Figure 11a. As for the Azure price plan, there are two different options for how the user can be billed. It is either the consumption plan that has been used in this project or app's service plan. In the app's service plan a function runs dedicated to a VM, so the benefits of paying per use are gone and therefore I excluded it in this thesis. With Azure Functions when executing a function, it can not exceed $1024MB$ in memory and the function has a time limit of ten minutes. The user is billed for the nearest $100mS$ and at least $128MB$ in memory, so it is important to optimize the function and do not use to small functions. To run a function for ten minutes, you must manually change it in the function.json file, since the update just arrived. In Azure Functions, a user can create a Function App that allows the user to scale up to 200 instances. Table 1 shows the consumption plan for Azure Functions, [1].

Table 1: Azure Function consumption plan

	Price	Free grant (per month)
Execution time	\$0.000016 /GB-s	400 000 GB-s
Total # of runs	\$0.20 per million executions	1 million executions

One thing to mention is that storage is not included in the free grant. Azure offers new customers a free version where the user can test Microsoft Azure with 1600 in fictional currency, [1].

Amazon Web Services, AWS, is currently the largest supplier of FaaS called AWS Lambda, Figure 11b. If AWS Lambda is compared with Azure Functions, the price plan is almost identical. AWS Lambda has the same minimum allocation area as Azure functions at $128MB$ but has triple in maximum memory use. In AWS Lambda you can create functions that require a memory use up to $3008MB$, with $64MB$ in increments and the maximum runtime is five minutes. In Table 2 shows the price plan for AWS Lambda, [14].

Table 2: AWS Lambda price plan

	Price	Free payment (per month)
Execution time	\$0.00001667 /GB-s	400 000 GB-s
Total # of runs	\$0.20 per million executions	1 million executions

The last cloud service I have evaluated is Google Cloud. The functionality of FaaS was developing by Google in 2017 and it is not as advanced compared to the other two actors. Google Cloud Function, Figure 11c, is built so that it is very cheap to execute a function with regard to time but little more expensive for every million executions. A function is allowed to run for 540 seconds with maximum 1000 deployed functions in each project. Each function is allowed to have a maximum size of $100MB$, which is much lower than the other actors. How this affects the user will be evaluated and presented in the results section, [10].

Table 3: Google Functions price plan for FaaS

	Price	Free payment (per month)
Execution time	\$0.0000025 /GB-s	400 000 GB-s
Total # of runs	\$0.40 per million executions	2 million executions

FaaS is growing and developing all the time and more companies have started to offer FaaS. It is important to remember that it is still in develop-

ment, there are only a few cloud computing service that are offering FaaS on a large scale. IBM offers an open source system for FaaS, called IBM Cloud Functions (based on OpenWhisk), which the user has the ability to use it in the cloud or on premises. Brief facts about IBM Cloud Functions also offers 400 000 free GB-s, cost $0.000017 per execution and rounded up to the nearest $100mS$. A function can run up to five minutes but can only use $48MB$ in memory. OpenWhisk has not been taken into account in this project due to time constraints, [11].

3 Method

In this section, I go through how FaaS have been added and tested to different architectures.

3.1 FaaS added to a monolithic- and microservice architecture

To evaluate FaaS and its advantages in various architectures, I began my thesis by learning the basics of monolithic- and microservices architecture. I also needed to have knowledge of API and JSON to understand how FaaS should communicate, set bindings and configurations. I used a web API to communicate with my microservice and my FaaS and the Web API:s "get"-method to receive the information from the FaaS, Figure 13. In API, I decentralized my JSON-file containing the information needed to trigger my FaaS through an Http-request. FaaS has its own function.json file, Figure 6, where all configurations and bindings were determined for FaaS. Figure 12 shows how I tested how to attach FaaS to a monolithic and microservices architecture. I also evaluated whether FaaS could be executed in a container.

FaaS added to a Monolithic- and a Microservice architecture

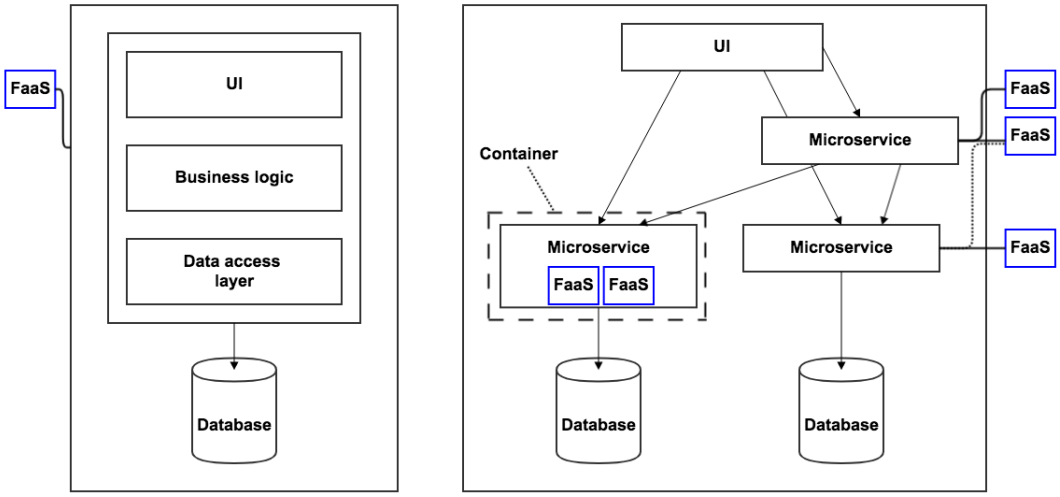


Figure 12: Shows a general example of different approaches I've used to add FaaS to a monolithic and microservices architecture

In Figure 13 you can see a part of the code to show how I communicate and execute my FaaS. It is done via a Web API and its "get"-method, where I call my FaaS via an Http-request. In order to send the information needed for my calculations in FaaS, I create a JSON file. I deserialize the file to make it manageable and able to save it. With this information, I can then activate my FaaS which required in this case, a total debt, interest rate, user ID, etc. FaaS returns a response in the form of a string that I then receive and calls "responseString". Model-View-Controller, MVC, HomeController, Figure 14, was used to determine what response a user receives when it has made a request for eg. a microservice. In this case, the response is my FaaS via a Web API, so now FaaS becomes part of the service.

Web API Controller

```
private static readonly HttpClient client = new HttpClient();


[HttpGet]
public async Task<string> ExMethodAsync()
{
    StreamReader r = new StreamReader("JsonFile.json");
    string json = r.ReadToEnd();
    Jsonpersonex account = JsonConvert.DeserializeObject<Jsonpersonex>(json);
    var responseString = await client.GetStringAsync("https://faastestandreas.azurewebsites.net/api/products/"
                        + account.Debt[0] + "/" + account.UserID[0] + "/" + account.Rate[0] + "/" + account.Days[0]
                          + "?code=x4zpa6kBVr6JBH434ukMc02llBW64upVUCQ9uMesKLUaRIGUmNUaqQ==&name=" + account.Name[0]);
    return responseString;

}
```

Figure 13: An example of how I used the Web API:s "get" method to communicate with my FaaS

MVC HomeController

```
[HttpGet]
public IActionResult About()
{
    HttpClient client = new HttpClient();
    var result = client.GetAsync("http://api/api/values").Result;
    string text = result.Content.ReadAsStringAsync().Result;
    ViewData["Message"] = text;
    return View();
}
```

Figure 14: The Model-View-Controller that determines what kind of a response a user receives when making a browser request

In order to be able to execute FaaS in a docker container, a Dockerfile is required with a base image with a FaaS runtime environment e.g. Azure Functions, Figure 15. By adding a base image, a container can make an almost identical copy of the runtime environment. This makes it possible to execute this function at any time and anywhere, using the same runtime environment as Azure Functions uses in the public cloud. The Dockerfile can have many different layers depending on what the developer needs. However, when the runtime environment is determined, the other layers work in the same way as explained in the theory section. Thus, by using the Azure Function runtime environment, the user can develop FaaS in a container with the same environment as the Azure Functions platform provides. Figure 16 shows an example of a docker-compose file where two FaaS are created and added to a container. Both the FaaS have their own Docker files based on their requirements and we can access the functions via the container access port 8080. The docker-compose file allows multiple FaaS to be added in the same container.

Dockerfile

```
1  FROM microsoft/azure-functions-runtime:v2.0.0-beta1
```

Figure 15: An example of how a container creates an almost identical runtime environment like Azure Functions has, in order to allow FaaS to run in a container

Docker-compose file

```
1  version: "3"
2
3  services:
4    faas1:
5      image: faas1
6      build:
7        context: .
8        dockerfile: Dockerfile
9
10   faas2:
11     image: faas2
12     build:
13       context: .
14       dockerfile: Dockerfile
15
16     ports:
17       - 8080:80
18
19
```

Figure 16: An example of how two FaaS can be executed in a container with all its dependence, port 8080, makes it possible to access the functions contained in the container

When evaluating FaaS in various architectural environments, all functions were created with Azure Functions. I created my functions both directly on the Microsoft Azure Cloud Platform and locally on my computer. I downloaded the Azure Functions package to run it locally, I wrote my functions and microservices in Visual Studio using the programming languages *C#*. Azure Functions is not the cheapest or better than any other supplier but combined with Visual Studio and an easy-to-use free account on Azure, it became my choice. The open source services Docker has been used to encapsulate services and functions in containers.

4 Results

In this section, I will review how FaaS can be applied in different architectures and its advantages. I will also present cost estimates and a comparison between the FaaS suppliers.

4.1 Evaluation of FaaS used in different architectures

In this master thesis, I have studied the use of FaaS in various architectural environments. The first test I did was to see if it was possible to add FaaS to a monolithic architecture and to understand FaaS advantages and opportunities. My second test was to create functions where they would be added to an architecture where the application runs on a VM. The functions also run on a VM, this has been evaluated to understand why the benefits of FaaS are needed. The third approach was to use FaaS together with a microservice architecture, where the entire application runs in the cloud. The fourth and final evaluation was to see if it was possible to execute FaaS locally with a Docker container. The main goal of this evaluation has been to see if it possible to add FaaS to an architecture to handle scaling problems.

4.1.1 FaaS added to a Monolithic Architecture

The first evaluation of FaaS was to add it to an existing monolithic architecture that was executed locally. For example, if an application reaches a bottleneck in the infrastructure, execution of the code can be delayed. One idea to overcome this problem is to see if the FaaS can be added as an addition to a monolithic architecture to handle the larger information flow. The approach with FaaS, in this case, was to run the function in the public cloud and use it when scaling was needed. I assumed hypothetically that the bottleneck occurred when users needed some form of calculation. I used an Http-request as an event to trigger my FaaS and the function was executed whenever a calculation was required. The function receives all the information needed and returns a value in the form of a string. In Figure 17 you can see an architecture of how FaaS can be added to a monolithic architecture. It is an easy and fast way to use FaaS, where you can replace functions in the application with FaaS or use it for scaling. With this approach, the developer saves memory and money by letting the cloud provider take care of the functions and all of its underlying complexity. An easy and fast way to extend an application and make it more efficient and handle bottlenecks without changing too much.

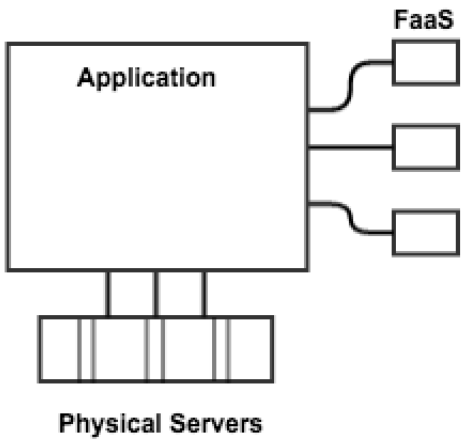


Figure 17: An architecture where FaaS runs in the cloud and is added to an application based on monolithic architecture

4.1.2 Architecture where the application and functions runs on VMs

The next approach was to create functions for an application running on VMs. This was evaluated to create an understanding of why FaaS and its functionality are needed. The functions were also dedicated to a VM and the functions were executed when needed. Because VMs are based on physical servers, there are still limits to how much space an application may use. However, when the functions are not needed, the VM can shut down and release the memory to other areas of the application. When an application reaches a bottleneck in the infrastructure, there are still limitations on how many functions can be created on a VM to handle all the information. Therefore, I understand that an application receives a certain amount of server, CPU, memory, etc. from the physical servers and when scaling is needed there are several issues that may occur. Regardless of whether an application runs on physical servers or VMs, the same problem will occur when scaling is needed. Thus, to understand the FaaS benefits and to continue with the thesis, I needed to understand the concept of how an application can be built and its restrictions.

Architecture where the application and functions runs on VMs

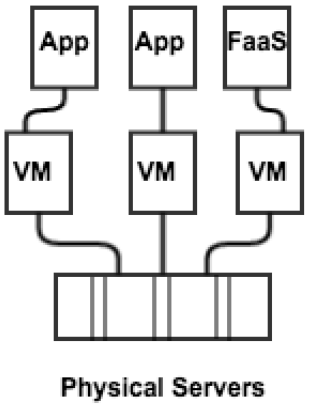


Figure 18: An architecture where an entire application runs on VMs, it was evaluated to understand the issues that may occur

4.1.3 FaaS added to a Microservice Architecture

The third approach I tested was adding FaaS to the application based on a microservice architecture where each service was encapsulated in a container. The containers are lightweight and applied to VMs, executed in the cloud. Here I have added FaaS as an add-on to an application to show that it is possible to add FaaS without making any major changes in the architecture. This can be used for existing applications, but also when building new ones. I will not focus on why the microservice architecture is beneficial, as explained in the theory section, but instead focus on why FaaS is a good addition. As mentioned in the first case of monolithic architecture, this is a perfect way to outsource some of the underlying complexity. By calling a FaaS with an Http-request via a Web API connected to a microservice, I could easily replace functions running on the VM with FaaS. In Figure 19 you can see an illustration of the architecture explained above. The result of this survey shows the simplicity of adding a FaaS to an existing microservices architecture, to both handle scaling and make the application use less memory. This leads to both cheaper and faster applications.

FaaS added to a Microservice Architecture

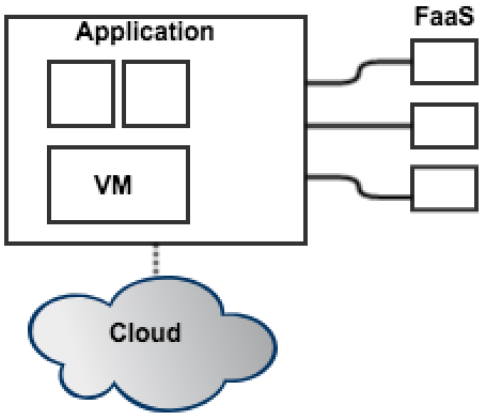


Figure 19: An example of an architecture about how FaaS can be added to a microservices architecture when FaaS is running in the cloud

4.1.4 FaaS encapsulated in a container and added to a Microservice Architecture

The last approach was to try to execute FaaS in a Docker container to make it more flexible, smaller and easier to develop and use in production. The containers containing FaaS are possible to run on the same VM as microservices that are also encapsulated in containers. The idea is to create a multi-FaaS container that starts when needed. With this architecture, we can now use FaaS no matter how the customer wants to run their application. By using FaaS in a container, it is now possible to execute FaaS directly on the same VM as the application is built without using the public cloud. Therefore, with this flexibility, you can develop and test all FaaS locally and move the container without any changes.

FaaS encapsulated in a container and added to a Microservice Architecture

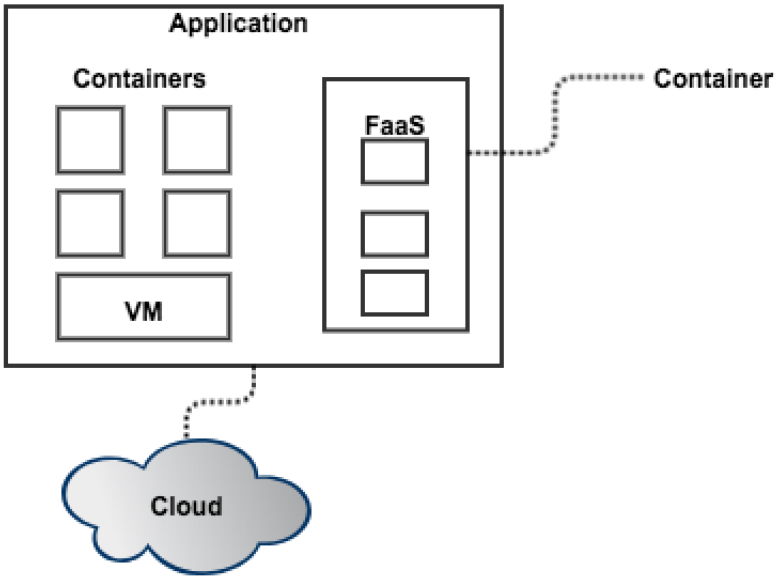


Figure 20: An example of an architecture on how FaaS encapsulated a container can be added to the microservice architecture, the application can also be run locally due to the use of a container

4.2 Benefits with FaaS compared to physical servers and VMs

There are two main aspects to why companies would want to improve their application. First and foremost as I explained in the previous section, it is to make the application more efficient. The other aspect is the cost effects. In this section, I have simulated the differences between when FaaS is running in the public cloud compared to when the same functions are executed directly on physical servers or on Virtual Machines.

In the following example, I simulate a smaller example where I run ten million invocations and each run takes sixty seconds to run. The function requires $512MB$ in memory and I will calculate what the monthly cost will be for each case.

I have used Azure Functions when calculating the price of using FaaS and I start by calculating FaaS price per month. The total duration will then be six hundred million seconds, as can be seen in Table 4.

Table 4: Resource consumption (seconds)

Executions	10 million
Execution duration (seconds)	x 60s
Total duration (seconds)	600 million s

Each function demands a memory for 512MB and if we convert it to GB we get $\frac{1}{2}$GB. If we multiply the resource consumption in GB with total execution time we receive the resource consumption in GB-s.

Table 5: Resource consumption (GB-s)

Resource consumption converted to GBs	512/1024
Execution time (seconds)	x 600 million s
Total resource consumption (GB-s)	300 million GB-s

As mentioned in the theory part, Azure Functions provides the user with 400 000 GB-s for free every month. 300 000 000 − 400 000 = 299 600 000 GB-s. For the calculation of the total cost, Swedish crowns have been used, where \$0.000016 ≈ 0.000126kr and \$0.20 ≈ 1.575kr.

Table 6: Total cost for GB-s

Total cost (Kr)	299600000 GB-s x 0.000126 kr/GB-s = 37749.6$kr/month$

I have calculated the total cost of GB usage each month, now we want to add the monthly resource consumption cost. Monthly resource consumption cost is 10 000 000 − 1 000 000 (free) = 9 000 000 executions that will be charged. Each million execution cost 1.575kr, so 9 times 1.575 equals 14.175kr. By using Azure functions with ten million executions and 512MB in memory the total cost is 37905.52$kr/month$.

If we want to use FaaS as a function running on a VM instead, I have made some basic calculations to show you the differences.

At IBM Softlayer you can order Virtual Machines with your own customizations, I have done this to simulate the cost differences between the three approaches, [25]. As can be seen in equation (1) one can see how much memory the function will need when they run simultaneously. Each function allocates 512MB so the server needs to at least have 115GB in memory if we round up the server will have 128GB in memory. On a VM there needs to be storage, operating system, HyperVisor, maintenance, etc. Two VMs are necessary in case of system failure which means that two VMs with 128GB

memory with advanced monitoring and the extra features and with unlimited bandwidth cost $69909.93kr/month$.

If we compare FaaS and functions running on physical servers we get following calculations.

I have taken a rather small example, so I have been able to compare that to if I would purchase my own physical servers. First of all, I need to purchase two separate servers in case of system failure to use one as a backup. If I buy and use two HPE ProLiant ML150 Gen9 servers where each server costs 14 $495kr$, the total price for both will be 28 $990kr$, [7]. The servers only have $16GB$ in memory, which means I have to buy extra memory and also storage to allow the operating system to be installed. If I use exactly the same number of invocations and runs, which is ten million invocations and each function is running for sixty seconds. In equation (1) I calculate how many seconds there are in the month and it will run 300 million executions every month, so every second, the server must have at least 115.74 GB in memory.

$$60x60x24x30 = 2\ 592\ 000s, \qquad \frac{300\ 000\ 000GBs}{2\ 592\ 000s} \approx 115.74GB \tag{1}$$

To the server, there needs to add 120 x 2 GB in memory. If I add the memory, I need 15 items of 8GB memory in each server, which will cost $46350kr$ where each memory cost $1545kr$. I will also need some storage to have my operating system on, etc. I have chosen storage of $150GB$, which some may argue that it is too much but the cost does not affect the total so much. Two number of storage cost together $2398kr$ and the total cost is $77738kr$.

Table 7: Comparison between FaaS, VM and Physical Servers

	FaaS	VM	Physical Servers
Executions	10 million	10 million	10 million
Duration (seconds)	x 60s	x 60s	x 60s
Memory	512MB	512MB	512MB
Total cost	37906kr	69910kr	77738kr

The total cost of physical servers is more expensive and I have not even included electricity costs, storage for the servers, maintenance etc. There will be even bigger problems and cost differences if an application is increasing in memory and needs to scale. For FaaS, you do not need to change anything because the cloud supplier takes care of the scaling. With a VM you need to

create a new one or increase the existing VM that costs more money. With physical servers, you need to purchase new components where updates and maintenance are needed, which will cost even more money. Thus, FaaS is the cheapest choice, but the importance of simplicity and efficiency is the key factor.

4.3 Comparison between the consumption plan between the three largest FaaS providers

In this section, I present the calculations of the price differences between the FaaS providers. I will show an example how the calculations have been performed and also illustrate what happens when different parameters are changed. For the calculation example, Azure Functions, AWS Lambda, and Google Functions have been compared. The functions are simulated to be executed ten million times, each invocation takes sixty seconds and allocates $512MB$ memory. Table 8 shows the first step, how to calculate the total duration in seconds.

Table 8: Resource consumption (seconds)

	Functions
Executions	10 million
Duration (seconds)	x 60s
Total duration (seconds)	600 million s

With the total duration, we can then calculate the total consumption in GB-s for each cloud supplier. Since the total consumption is the same for all three, I have made a column for the calculations.

Table 9: Resource consumption (GB-s)

	Functions
Converted to GB-s	512/1024
Execution time	x 600 million s
Total consumption	x 300 million GB-s

All three FaaS suppliers allow the customer to use 400 000 GB-s free of charge, leaving the total GB-s charged with 299 600 000 GB-s. However, each platform has a specific cost for every GB-s. All calculations have been made in Swedish crowns, SEK, and in Table 10 I have converted USD to SEK in today's course, (7 Dec 2017). The cost of each GB-s in SEK has been

rounded up to nearest fifth decimal, it has been done to make the calculation as fair as possible.

Table 10: USD converted to SEK, (7 Dec 2017)

FaaS providers	Azure Functions	AWS Lambda	Google Functions
Cost for each..			
GB-s ($)	$0.000016	$0.00001667	$0.0000025
GB-s (Kr)	$0.00013kr$	$0.00014kr$	$0.00002kr$
million of runs ($)	$0.20	$0.20	$0.40
million of runs (Kr)	$1.575kr$	$1.575kr$	$3.150kr$

Because the biggest memory usage for Google Functions is that each function may not exceed 100MB. Thus, we need to create six functions to cover the memory of 512MB, which can be seen in the Table 13.

Table 11: Total cost for GB-s with Azure Functions

Total cost (Kr)	299600000 GB-s x $0.00013kr$/GB-s = $38948kr/month$

Table 12: Total cost for GB-s with AWS Lambda

Total cost (Kr)	299600000 GB-s x $0.00014kr$/GB-s = $41944kr/month$

Table 13: Total cost for GB-s with Google Functions

Total cost (Kr)	299600000 GB-s x $(0.00002\ x\ 6)kr$/GB-s = $35952kr/month$

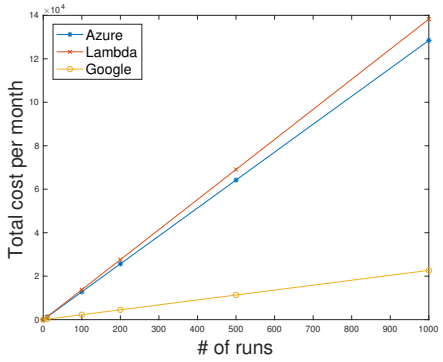
The number of executions that will be added to the monthly cost will be 10 000 000 − 1 000 000 (free) = 9 million executions for Azure Functions and AWS Lambda. For Google Function, there will be 10 000 000 − 2 000 000 (free) = 8 million executions. The cost of each million run is shown in Table 10. For Azure Functions and AWS Lambda, the cost of running a function for 9 million times is, 9 x 1.575 = $14.175kr$ and for Google Functions it is 8 x 3.150 = 9.45. We have to remember that we needed 6 times as many functions with Google Function, therefore we need to multiply 9.45 with 6 which equals $56.7kr$. Table 14 shows the total cost for the three FaaS suppliers for this particular example, rounded up to the nearest integer.

Table 14: Total cost

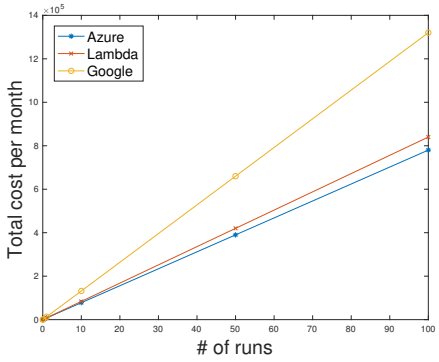
FaaS providers	Azure Functions	AWS Lambda	Google Functions
Total cost (Kr)	$38962kr$	$41958kr$	$36009kr$

Through this example, we can note that Google Functions is the cheapest option, however, this is not the entire truth. Depending on how much memory a function requires, runtime or number of executions, the result will vary. In Figure 21 shows the results where I have done the same calculation but changed different parameters.

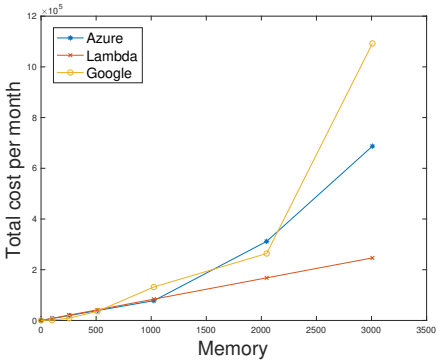
Comparison of the platform providers



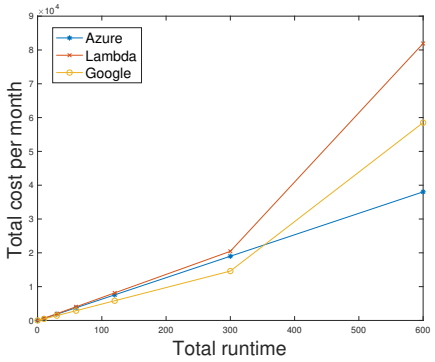
(a) Total cost per month as a function of number of executions, 100MB 10s



(b) Total cost per month as a function of number of executions, 1024MB 60s



(c) Total cost per month as a function of memory use for each function, 60s 10million



(d) Total cost per month as a function of total runtime, 512MB 10million

Figure 21: The graphs (a-d) indicate that depending on the type of functions you want to put in a FaaS, you must select the cloud service that best suits your requirements

As Figure 21 demonstrate, using FaaS does not mean that everyone that provides FaaS gives the same outcome of your function. Figure 21 shows four different graphs where each curve represents a FaaS provider. In each graph, I have changed different parameters to show some examples of the difference, but you can do endless many different analyses.

In Figure 21a I have the total cost per month as a function of a number of executions. The memory requirement for each function has been 100MB, which is Google Functions maximum memory for a function, where each function runs for ten seconds. The result shows that each curve is linear and that in this case, Google Function makes the best choice.

Figure 21b shows the total cost per month as a function of a number of executions. What I am showing here is that only by making the functions larger, (1024MB) and where they will run for 60 seconds makes a big difference compared to Figure 21a. By making these, quite small, changes now Google Function is by far much more expensive to use.

Figure 21c shows an outcome where I have determined the number of executions and time, a number of executions are equal to ten million and time for each invocation is equal to sixty seconds. Although the AWS Lambda and Azure Function have similar consumption plan, they have some differences in memory and runtime. In this case, because AWS Lambda allows each function to use 3008MB in memory, Lambda becomes much cheaper to use the greater the memory required by the function. Of course, when a function requires more memory needs, Azure Function and Google Function need to create more functions to achieve the same memory requirements.

Finally, Figure 21d displays the total cost per month as a function of total runtime. Here I have specified that each function requires 512MB and a total number of executions to ten million. Since all cloud providers allow every function to run for at least five minutes, 300 seconds, there are no major differences in the beginning. The longer a function runs the bigger the differences will be. Azure Functions allows each function to run up to ten minutes, 600 seconds, where the other two options need to adjust the number of functions to achieve the same total time required.

5 Discussion

Nowadays, several companies want to move their application to the cloud to make it both more cost-effective and efficient. One common problem is that many people move the entire application directly to the cloud. A problem may occur because now the application is not optimized to execute in the cloud and may even become even more expensive. It is crucial to first and foremost build an application that is suitable for performing in the cloud. By using this paper survey, many companies can improve their applications with their own customization. If some companies have scaling problems but still want the application to be executed in their own data center, they can simply add FaaS that is executed in the public cloud and save memory, money, time etc. As the result shows, FaaS is very flexible and can be used in many different ways. From just using FaaS to existing applications to handle a large flow of information to apply FaaS in a container and run locally, developers will have the opportunity to improve an application in the best possible way.

FaaS can also be used to move functions that run on physical servers or virtual machines. If you have functions that run only a couple of hours a day or the function that runs on a schedule, you could instead let them run as a FaaS. In other words, a function that does not run for twenty-four hours, seven days a week can instead be performed in a FaaS to save both money, but maybe even more importantly, space on the servers.

I have evaluated how to improve cost and efficiency, but one of the most important aspects is how security can be ensured when using FaaS in production. First of all, we have to decide how sensitive information we want to send to FaaS and where in the production the bottleneck is. One way to overcome that sensitive data is leaked is by using UUID (Universal Unique Identifier), where you send out a random number linked to, for example, an ID number. If you want to send out e.g. An ID number to a FaaS, you should send out a random number instead associated with the ID number. Certainly, there is still a risk that someone could figure out which random number is associated with which ID number. However, if we combine UUID with a time stamp so that it will never generate the same number twice, it would be almost impossible to figure out what information it contains. One thing worth mentioning is that a FaaS only runs for a short time and then ”dies”, so the time window for hacking a FaaS is very small. The reason why FaaS is so advantageous is that it is flexible. The result shows that FaaS can be used in many different ways and architectures. If a company is determined to execute an application in its own data center, FaaS can be designed to run in containers or directly on a VM. Thus, the company is responsible for

the security together with the remaining part of the application.

As a developer, it is important to be able to monitor an application as well as possible. When Faas is used, it is very likely that many FaaS will run simultaneously. It is then important to find the best and most effective way to monitor FaaS in order to use it in production. I put a version on each FaaS so I know which version I am using. The semantic version can be used, where a version consists of three numbers, [28]. Major.Minor.Patch and depending on what you change, change one of the numbers. Major is when you change in API, Minor is when you, for example, add a functionality and Patch is when you make smaller bug fixes. Developers need to know the version to keep track of which one to use.

When a version is set on FaaS and I apply it in a container, I need a good system to handle a containerized applications. A good open source application for managing an application is Kubernetes, where you can distribute, maintain and scale a containerized application. Kubernetes gives you the ability to run your application anywhere, local, hybrid or in the cloud, it is a system that provides an infrastructure for your containers, [12].

FaaS is a new addition to cloud services and it is evolving every day. By just comparing FaaS with VMs and physical servers, one can see that the benefits and opportunities are huge. FaaS is cheaper, but the simplicity with FaaS is an important factor to take in consideration. When scaling is needed with FaaS, the cloud service will handle it for us, within the limits of how much memory and runtime the function may have. While dealing with VMs and physical servers, one has to take care of it by themselves, which takes both time and money. Thus, more focus can be applied in other areas over the time saved by using FaaS.

The two most important aspects of the evaluation of FaaS are to make an application more effective and of course cheaper. FaaS undoubtedly gives the developer the opportunity to achieve both. However, it is very important to consider what Figure 21 shows that you have to make your own calculation to understand which FaaS platform provider fits your function as optimally as possible.

6 Conclusion and Future Work

6.1 Conclusion

My conclusions from this thesis is that FaaS is a good addition to both existing applications and in new applications architecture. Due to its flexibility, FaaS can be used in many different architectures. It can be used where there is, for example, bottlenecks in the infrastructure or just to use its cost effectiveness. However, just because it is a good addition, it still needs to be calculated which FaaS provider best suits your functions.

FaaS is in the construction phase and I am sure it will have a big impact on how applications are developed in the future to become even faster and cheaper. Quite a few companies offer FaaS, as soon as it will be used on a larger scale, I believe that even more companies will both use it and start developing it.

By evaluating how FaaS can be used in different architectures and evaluating its behavior in different environments, I conclude this conclusion. FaaS is a very flexible addition and can be tailored to customer requirements. I think FaaS will be widely used in the near future.

6.2 Future Work

A question for this master thesis is whether it is possible to combine FaaS with machine learning? For example, if I create a container that contains a large number of FaaS, can the use of machine learning help you to always find the cheapest supplier? Could this make FaaS even more beneficial, by always finding the cloud service that best suits my function?

I have made some discussions on how to set the version and how to protect data. But everything must be tested practically before it can be used in production. One must also find an optimal way to monitor their functions as well as possible.

7 Acknowledgments

First I want to express my appreciation to my Supervisor Torbjörn Lundmark and Tieto [26], who have given me the opportunity to do my master thesis with them. Extra appreciation of Torbjörn, who has guided and inspired me throughout the thesis.

To Andreas Hellander, who has been my subject's reviewer, a great appreciation for help with review and questions.

References

[1] Microsoft Azure. Introduction to azure functions. Retrieved 29 October 2017.

[2] Danny Bradbury. Microservices: Small parts with big advantages. 2 2016. Retrieved 30 October 2017.

[3] Mohsen Mosleh Kia Dalili and Babak Heydari. Distributed or monolithic? a computational architecture decision framework. 8 2016.

[4] Docker. Docker overview. Retrieved 5 Januari 2018.

[5] Docker. Get started, part 2: Containers. Retrieved 3 Januari 2018.

[6] Docker. What is docker. Retrieved 29 October 2017.

[7] Dustin. Dustin physical server. Retrieved 20 December 2017.

[8] ecma International. The json data interchange syntax. 12 2017. Retrieved 20 October 2017.

[9] Martin Fowler and Jamses Lewis. Microservices. 3 2014.

[10] Google Cloud Functions. Introducing google cloud functions. Retrieved 20 December 2017.

[11] IBM. Ibm cloud functions. Retrieved 5 Januari 2018.

[12] Kubernetes. Production-grade container orchestration. Retrieved 10 Januari 2018.

[13] Paolo Di Francesco Patricia Lago and Ivano Malavolta. Research on architecting microservices: Trends, focus, and potential for industrial adoption. 2017.

[14] Amazon Lambda. Introducing aws lambda functions. Retrieved 29 October 2017.

[15] Luke Marsden. The microservice revolution containerized applications data and all. 5 2015.

[16] Peter Mell and Timothy Grance. The nist definition of cloud computing. 9 2011.

[17] Ima Miri. Microservices vs. soa. 1 2017. Retrieved 10 Januari 2018.

[18] MuleSoft. Microservices vs monolithic architecture. Retrieved 5 Januari 2018.

[19] Shahir Daya Nguyen Van Duy Kameswara Eati Carlos M Ferreira Dejan Glozic Vasfi Gucer Manav Gupta Sunil Joshi Valerie Lampkin Marcelo Martins Shishir Narain and Ramratan Vennam. Microservices from theory to practice. 8 2015. Retrieved 5 Januari 2018.

[20] Claus Pahl and Pooyan Jamshidi. Microservices: A systematic mapping study. 5 2016.

[21] Chris Richardson. Pattern: Api gateway / backend for front-end. Retrieved 30 October 2017.

[22] Chris Richardson. Introduction to microservices. 5 2015. Retrieved 30 October 2017.

[23] Mike Roberts. Serverless architectures. 8 2016.

[24] Gregory M. Kurtzer Vanessa Sochat and Michael W. Bauer. Singularity: Scientific containers for mobility of compute. 5 2017.

[25] IBM Softlayer. Softlayer. Retrieved 20 December 2017.

[26] Tieto. Tieto.

[27] Matt Watson. What is function-as-a-service? serverless architectures are here! 5 2017. Retrieved 5 Januari 2018.

[28] Tom Preston Werner. Semantic versioning 2.0.0. Retrieved 10 Januari 2018.