Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Measurements of Function as a Service Platforms

Scientific Students' Conference

Written by
Hegyi Gellért István

Consultant
dr. Maliosz Markosz

October 28, 2018

Contents

Introduction

Serverless technologies are becoming more and more common and mature nowadays and it is safe to say that they will have a powerful impact on a lot of fields in computing. This paper will introduce the serverless paradigm, give a good sense about what are their advantages and disadvantages to the currently widely used solutions and how they compare in terms of performance.

The serverless, or in other words Function as a Service (FaaS) technologies, appeared on the market several years ago, but only nowadays started to get popular. This is partly because the performance and reliability of the early solutions were far from production grade, so it took some time, until the technology got mature enough, which was also helped by more and more competitors appearing on the market.

One of the most important aspect of serverless, that the developer does not need to care about the infrastructure of the application. The only thing that need to be taken into consideration is that a serverless application has to be stateless. This means that the program cannot preserve an internal state and rely on it on the application logic level. A serverless app instance can start, terminate and restart anytime and any number of time, so the developer has to prepare the code for it. This way all infrastructure operations are handled by the service provider and not by the client.

This technology also solves another problem of cloud-based applications: scalability. Stateless and container-based applications are relatively easy to scale horizontally, because it is easy to spin up multiple instance of the application on multiple machines and then directing requests to them evenly. The other advantage of serverless in terms of scalability is pricing, because this concept allows for a customer to pay only for the computation it consumed, without running servers for a long time and paying for unused time also. This makes it a lot more affordable option.[14]

Such a new concept also comes with the question, whether it can also provide the same performance as traditional servers. I will explores this question and problems related to it. By the end it will be easier to compare serverless approaches to non-serverless ones so one can make a better decision, when it is time to choose the right technology for a problem. In this paper I will focus on the existing popular solutions, how they need to be set up and used and also some basic measurements regarding their latency and computational performance.

Chapter 1

Serverless technologies

The most important underlying building blocks of current Serverless platform are containers. A piece of code that is run on a Serverless platform is called a function, hence the name Function as a Service (see the architecture 1.1). When someone creates a Serverless function it is registered on the platform and can be accessed in multiple ways. One of the most obvious one is through an HTTP endpoint. There is no dedicated server behind a function with a public IP address and port, so the function's endpoint is handled by some sort of load balancer (e.g. API Gateway), that can invoke the function.[12]

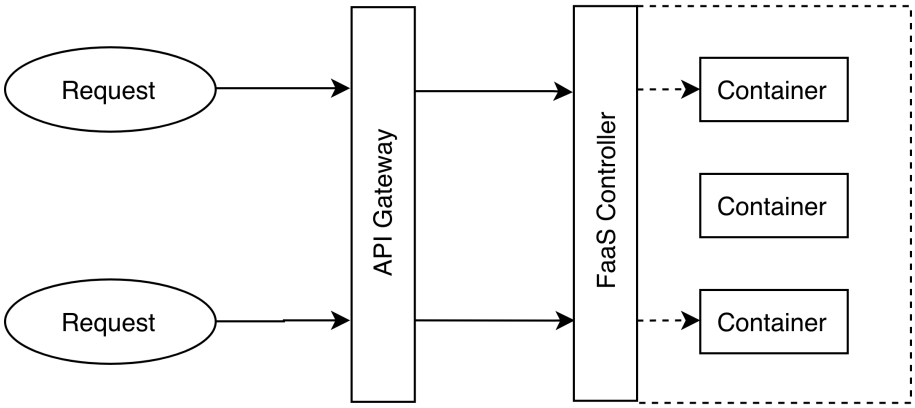

Figure 1.1: *Function as a Service architecture*

Invoking a function can mean two things. Either a new container needs to be spin up from its image, with its corresponding application code, or an existing container of the function is already running and it can be used right away. Starting a new container can take significant time, so in this case the latency is much higher. This is called the cold start time. When a function invocation is served the platform keeps that container running for some time (this is platform dependent), but after a while it terminates the container to release the allocated resources. This is the mechanism that allows a FaaS to achieve great scalability and to bill after actual usage.[21]

The first commercial Serverless solution was shipped by Amazon Web Services (AWS) in 2014, which is called Lambda. Since then many competitors appeared, but this is currently

the most popular option. A Lambda function can be invoked in three different ways: it can be called directly (invoke), with its unique identifier through a supporting framework, it can be called via an HTTP endpoint as mentioned above, or it can be called via triggers through an ecosystem, such as database insertion events.

After the initial success of Lambda a lot of big cloud providers started to roll out their own FaaS solution. These are being developed and spreading fast nowadays. Aside from these proprietary solutions, there are also public, open-source ones, but these are in even earlier stages. There are several online sites that enlist the serverless solutions (e.g., [18]). Relying on my experiences with such solutions, I drafted the following shortlist of the main solutions available today:

- Lambda: private, served by Amazon Web Services,[8]
- Azure Functions: private, served by Microsoft,[3]
- Google Cloud Functions: private, served by Google,[15]
- Bluemix OpenWhisk: private, served by IBM,[7]
- Fission: open-source, built on top of Kubernetes,[10]
- Gestalt: open-source, built on top of Kubernetes,[9]
- Kubeless: open-source, built on top of Kubernetes,[13]
- OpenFaaS: open-source, built on top of Kubernetes or Docker Swarm.[19]

During my work I could not cover all the serverless solutions due to lack of resources. I tried instead to select a subset of the above list instead, that is generic enough to give a comprehensive look about the currently available serverless solutions. It includes the most important public solutions, which are important for the production level implementations, and a feature-rich open source technology with large visibility in the open source community. Therefore my picks are: AWS Lambda, Google Cloud Functions, Microsoft Azure Functions and Fission.

1.1 AWS Lambda

Amazon Web Services was the first cloud provider to release a Function as a Service platform back in 2014, Lambda. It has the same basic underlying architecture as described before. In Lambda, one can define functions, that can be called with a set of parameter objects. At launch, only Node.js environment was available, but today more development environments are available, for example Go, C#, Java, etc.

As opposed to some other alternatives, Lambda does not provide lots of convenience features out of the box, such as HTTP endpoints for the functions, the developer has

to set this up manually. For a Lambda function one performance related measure can be configured: memory. There are a couple of different sizes available from 128MB to 3008MB, with 64MB steps. Though it is not officially documented, but it can be observed that higher memory sizes may also mean a stronger CPU, both in cores and in clock speed as well.

Configuring these settings can be quite tedious, therefore a number of tools have been created to help these processes. The most popular is the Serverless framework [ref], which lets developers create functions for many different platforms easily and consistently. For Lambda, it does all the configuration and deployment with a simple YAML descriptor file, through CloudFormation templates.

1.2 Google Cloud Functions

Google's Function as a Service solution is one of the newer ones. During the research period, for a long time is was in beta stage and was only available in one of Google's operating regions, in eastern US. However, in August 2018, it came out of beta and now available in multiple regions as well. Initially being in beta was a bit of a problem for the measurements, because all the other platforms have at least one region available in Europe.

Cloud Functions make it possible to set the memory size of the instances and also lately even the specific CPU types can be chosen as well, moreover functions get a URL to call them through HTTP by default. The Serverless framework also supports this platform, so its usage does not differ much from the AWS Lambda solution.

However, my personal experience using the two is that I had an easier time working with the AWS solution. This is mostly because the initial setup for Google Cloud is more cumbersome than for AWS and moreover it is not so well documented. The most important part of this is setting the security credentials right, which are needed to deploy the functions.

1.3 Microsoft Azure Functions

The Microsoft equivalent is almost exactly the same as the previous two ones, but relates more to Google Cloud Functions. It can be used with the Serverless framework, but it also caused the most pain in terms of usage. The authentication for the deployment was quite hard and tedious. Also the online management dashboard is the worst regarding user experience compared to the other solutions.

Regarding the configuration of the functions, we do not have that much options since here we cannot set the size of the memory or the type of the CPU we would like to use.

1.4 Fission

Fission is probably the newest technology among the platforms that are part of the measurements (around one year old) and it is also the only open source one, which means that the deployment of the Function as a Service is also our responsibility. It is based on Google's container orchestration technology, Kubernetes. Kubernetes is an open source solution to automatically deploy, manage and scale containers. Because the foundations of the FaaS methodology is also based on containers, it is quite obvious to use Kubernetes to implement such a mechanism.

To set up its environment was by far the biggest task compared to the setup of the others. This is in part because it is quite new, so its documentations are not well developed. To use it we need a fully functional Kubernetes cluster that I set up on the Google Cloud Platform. Then I had to install the Fission function management tool onto it. With help from the creators of the platform I could deploy my example application. It is a promising idea, but it is still needs work to be able to use it in production.

Chapter 2

Initial measurements

2.1 Performance factors

At first I looked through all the articles that are concerned with the performance of the FaaS platforms, which is not in high numbers because of the novelty of the technology and most of them are focused on one particular platform. Measurement of cloud services can take on many forms and lots of aspects can be observed that affects performance. For initial measurements I chose the two most important and simple to measure: latency and computational capacity.[5]

2.2 Measurement tools

I performed the measurements in Node.js environment, because nowadays it is the most popular thus all of the current FaaS solutions support it. I also used tools to do the measurements. Artillery is a command line tool written in JavaScript that helps to performs scripted latency measurements. We can define the duration and the number of calls to make per second. It also supports custom code execution to process the HTTP response, so it could be used in the calculation performance test too. Artillery can execute scripts written in YAML syntax and provides results about latency with 95 and 99 percentiles (see a configuration snippet in figure 2.1.)[16]

Deployment of FaaS functions differ on every platform and can be quite tedious. There are many tools to help developers with this and to unify the methods of function deployment. The most widely used is called the Serverless framework. It is a command-line tool and supports all the big cloud providers. It uses a YAML descriptor, serverless.yml, that contains every information about the function to be deployed (see figure 2.2.) It can create URLs to the functions automatically, and helps in managing credentials for different platforms.[11]

Almost all of the open source FaaS frameworks are built on top of Kubernetes, and Fission is no exception. After setting up a Kubernetes cluster in Google Cloud, we need to install

```
config:
  target: "https://latency.measurement.url"
  phases:
    - duration: 60
      arrivalRate: 50
  processor: './record-response.js'
scenarios:
  - flow:
    - get:
        url: '/'
        capture:
          - json: '$.timestamp'
            as: 'timestamp'
        beforeRequest: 'recordSend'
        afterResponse: 'recordReceive'
```

Figure 2.1: *Simple example of an Artillery config*

the Fission CLI with a package manager for Kubernetes, called Tiller. This sometimes can be tricky, because it is important to have a load-balancer in front of the cluster, with a public IP address, in order to access the deployed functions from outside.

Once we have the Fission CLI installed, it is easy to create functions. With the help of a couple of commands we can deploy functions to different endpoints (see figure 2.3.) There are multiple environments supported, Node.js is one of them. Important to note that Fission requires to specify the code of a function in a single file. This means a code bundler, such as Webpack[1] might be useful.

Creating a function with Fission on a Kubernetes cluster:

2.3 Environments

As mentioned earlier, the measured applications that were deployed on the observed FaaS solutions are written in Node.js environment. These are simple web servers that in case of the latency measurement sends back a response message immediately. In the computational capacity measurement application, it executes the multiplication of two randomized matrices, of which sizes are determined by a query parameter, thus making it possible to define tasks that are harder or easier to compute.

Besides the mentioned FaaS platforms it is also important to compare performance to more traditional approaches that are common nowadays. That is why I also created simple web servers in a single container. Comparing this with a FaaS result from the same provider can show the overhead of the serverless abstraction layer.

[1] https://webpack.js.org/, accessed 28-October-2018

```
service: load-test
provider:
  name: aws
  runtime: nodejs6.10
  region: eu-central-1
  memorySize: 1024
  timeout: 30

functions:
  calc:
    handler: handler.calc
    events:
      - http:
          path: 'calc'
          method: get
```

Figure 2.2: *Example for a serverless config*

```
$ fission route create --function hello --url /hello \
        trigger '5327e9a7-6d87-4533-a4fb-c67f55b1e492' created

$ fission fn create --name hello --code hello.js \
    --env node --executortype poolmgr
```

Figure 2.3: *Commands to create a Fission function and a HTTP route for it*

Measuring latency from a home network environment is not reliable, because there are a lot of outside factors that can affect the results. That is why I performed my measurements from an independent cloud provider, Digital Ocean[2]. This allowed to have consistent results that can show the real difference between solutions.

Another important aspect is performance inside a cloud provider. That is why I also performed latency measurements, the same as outside, from a cloud provider, calling its own FaaS implementation. I executed the scripts from a simple virtual machine on Amazon and Google.

2.4 Measurement types

2.4.1 Latency

For the latency measurement I deployed a simple web server that in response of a HTTP request, sends back a response message immediately (see a code snippet in figure 2.4.) Artillery makes it possible to create scripts that puts different amount of load on the measured endpoints.

[2]https://www.digitalocean.com/, accessed 28-October-2018

I created three different test cases, with three different amount of load (see Section 3.1 for details.) I ran these for the same duration, which is around 5 minutes. To have consistent and real results, I repeated the measurements multiple times and calculated average values from them. I used this technique at every measurement type.

The application code for the AWS Lambda function shows that the main control block is just a function:

```
const simpleResponse = require('./response')

module.exports.hello = (event, context, callback) => {
  const response = {
    statusCode: 200,
    body: JSON.stringify(simpleResponse('AWS Lambda')),
  }

  callback(null, response)
}
```

Figure 2.4: *Latency measuring function written in Node.js*

2.4.2 Computational performance

In the calculation measurement application, it executes the multiplication of two randomized matrices, of which sizes are determined by a query parameter, thus making it possible to define tasks that are harder or easier to compute (see a code snippet in figure 2.5.)

Similar to the latency test, I used three different matrix sizes to create different tasks. I ran these in the same fashion as in the previous test.

The application code for the AWS Lambda function:

```
const loadTest = require('./load-test')

module.exports.calc = (event, context, callback) => {
  const n = +event.queryStringParameters.n

  if (!n) {
    callback(null, {
      statusCode: 200,
      body: JSON.stringify({ message: 'Nothing' }),
    })
    return
  }

  callback(null, {
    statusCode: 200,
    body: JSON.stringify(loadTest(n)),
  })
}
```

Figure 2.5: *Computational performance measuring function*

Chapter 3

Measurement results

As discussed in earlier sections I performed the measurements in all of the environments, that is, between Digital Ocean and the measured providers and inside the providers. I run the tests for the different levels with the parameters seen in figure 3.1.

Measurement types	Duration	Load
Low	30 seconds	5 requests/sec
Medium	30 seconds	5-10 requests/sec
High	30 seconds	20 requests/sec

Figure 3.1: *Latency measurement types*

3.1 Latency

The visualizations show a CDF function, cumulative distribution function, where the X axis is the possible latency values and the Y is the cumulative probability of a given latency (some of the results with mean values can be seen in figure 3.2.)

Types	EC2	Lambda	Fission	Azure Functions	Cloud Functions
Low	25 ms	35 ms	15 ms	24 ms	126 ms
Medium	25 ms	34 ms	15 ms	23 ms	133 ms
High	29 ms	270 ms	15 ms	571 ms	176 ms

Figure 3.2: *95th percentile latency measurement results*

3.1.1 AWS Lambda

Looking at the results it shows that on low and medium load the performance of the function is stable and maintains acceptably low, similar average value (figures 3.3, 3.4, 3.5

and 3.6). However on high load the response time became less stable and relatively higher, which in a latency-critical application might not be acceptable.

3.1.2 Google Cloud Functions

The results of Google Cloud Functions shows somewhat similar forms than AWS Lambda, but with slightly higher average latency with a bigger deviation. On high load (looking at figure 3.8), we can see an even less stable performance and bigger average latency, than in the previous case. It is important to note that because Cloud Functions is still in beta phase, it is only available in one US region, which means that the requests travelled between continents and this affects the performance significantly.

3.1.3 Fission

Looking at the results of the Kubernetes-based Fission function, we can see drastically different, lower, values (figure 3.7). This can be because by deploying our own FaaS solution, we can skip otherwise necessary mechanisms, such as throttling large amounts of requests, for faster performance. Here, even the high load performed quite similarly as the other two.

3.1.4 Microsoft Azure Functions

Measuring Microsoft Azure Functions turned out to be hardest one of all the solutions, because the service itself is quite unstable. This means that when I tried to measure, it produced unexpectedly high latency and a lot of times the whole service was unavailable.

Because of these reasons I decided not to proceed with the measurements of Azure functions. Though it has to be mentioned that the platform is still in beta stage, so these issues can be resolved in the future and then it can be a viable option.

3.1.5 AWS Docker

As a reference I also measured a traditional web server running in a Docker container, and used it to benchmark the rest of the scenarios. This comparison is important because, it shows the overhead of the Function as a Service abstraction layer.

The results show that on average the latency is in the same order of magnitude as Fission, so significantly faster than the other solutions and somewhat similar to its counterpart, Lambda.

3.1.6 Grouped results

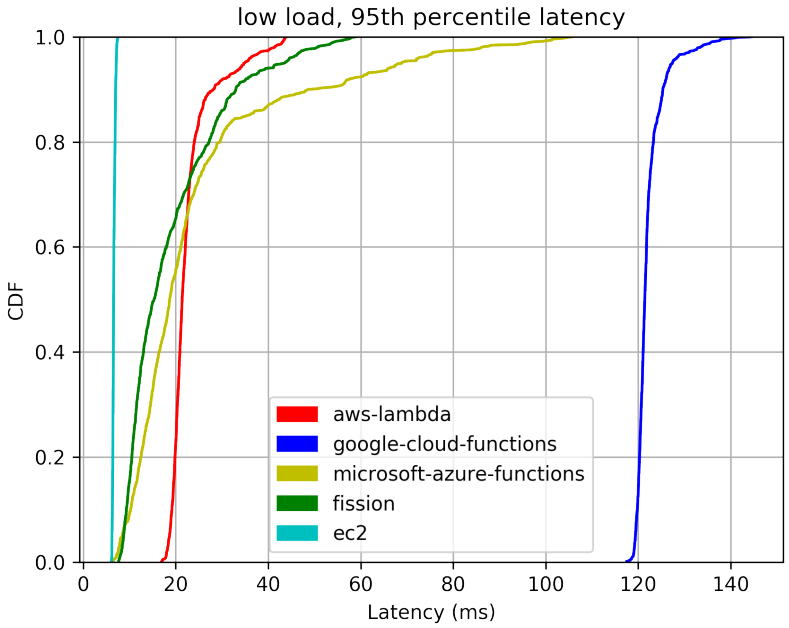

Figure 3.3: *95th percentile of first low latency measurements*

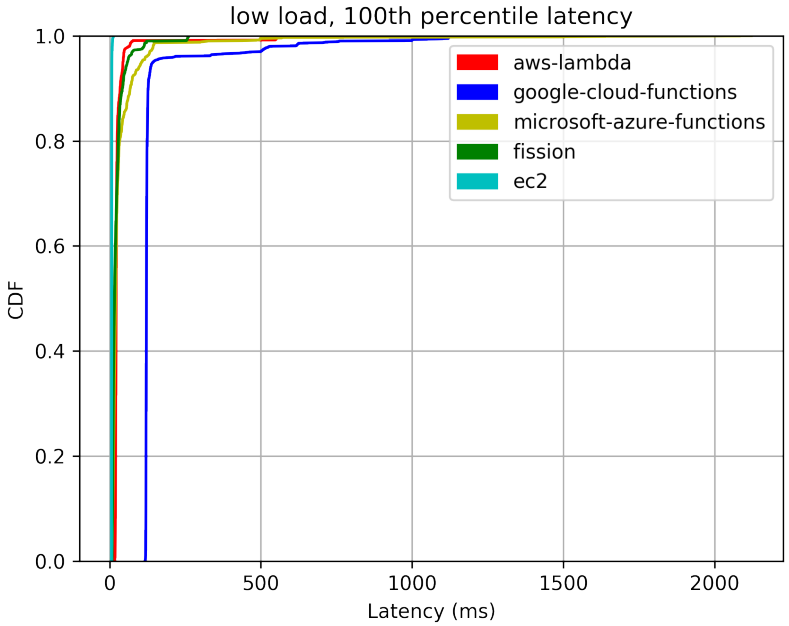

Figure 3.4: *100th percentile of first low latency measurements*

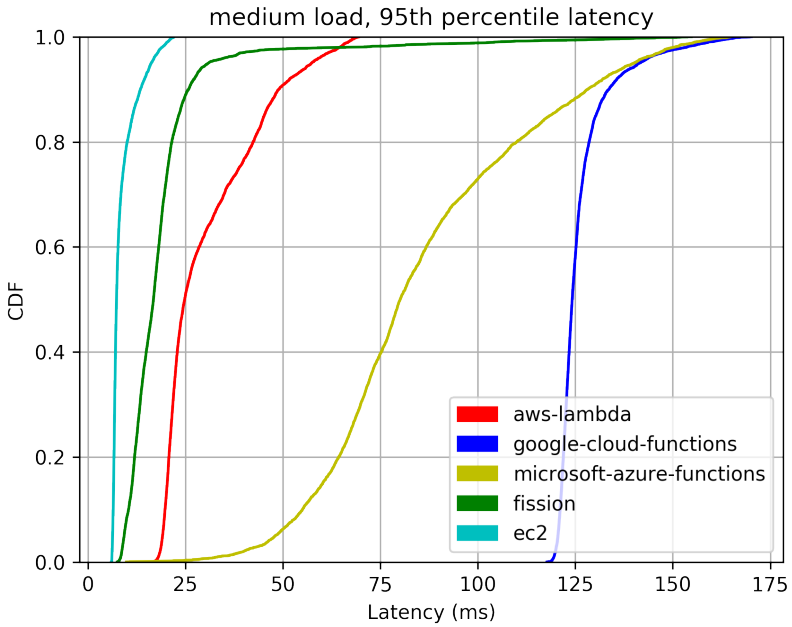

Figure 3.5: *95th percentile of first medium latency measurements*

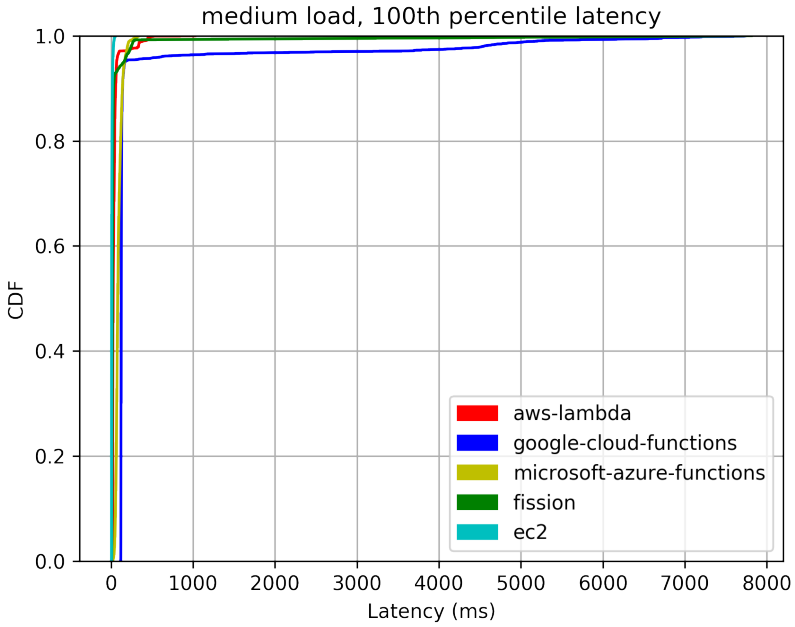

Figure 3.6: *100th percentile of first medium latency measurements*

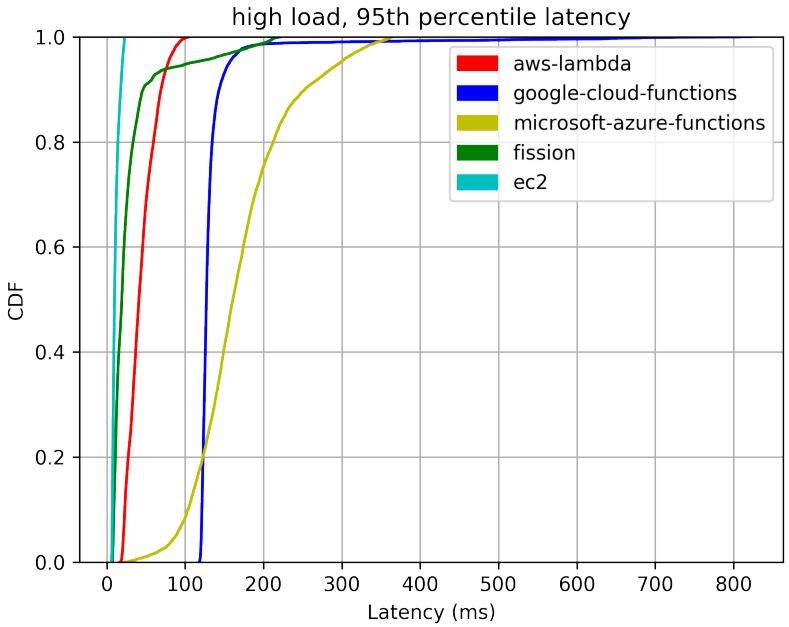

Figure 3.7: *95th percentile of first high latency measurements*

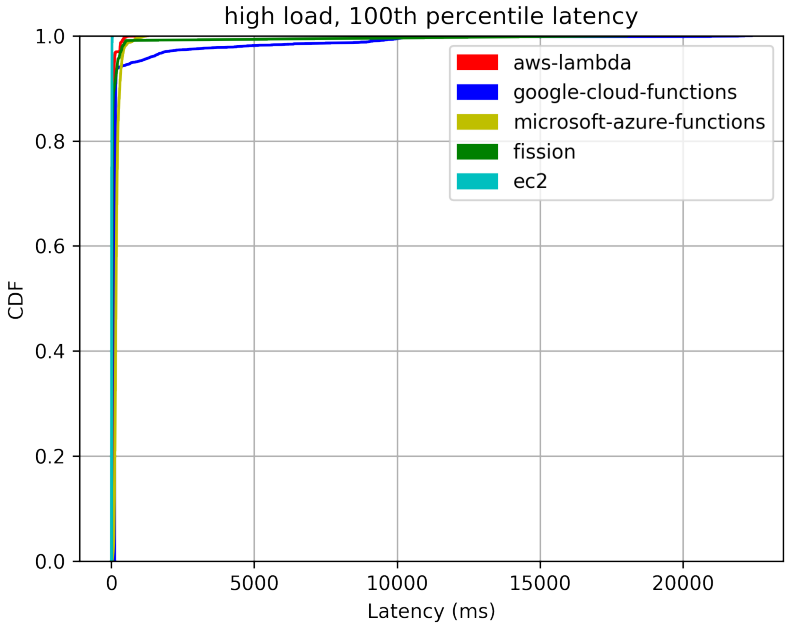

Figure 3.8: *100th percentile of first high latency measurements*

3.2 Computational performance

In this measurement it did not matter from where the request was made, because the test only measured operations made inside the FaaS container instance. The parameters are described on figure 3.9. The results are shown grouped by computational complexity, low,

medium and high. As mentioned earlier the performance of the used FaaS containers can be configured in a limited way through the assigned RAM size. Throughout the tests the platforms were configured to have the same performance as much as possible.

Measurement types	Duration	Load	Matrix size (nxn)
Low	10 seconds	1 requests/sec	50
Medium	10 seconds	2 requests/sec	200
High	10 seconds	4 requests/sec	400

Figure 3.9: *Computational measurement types*

3.2.1 Results

Low

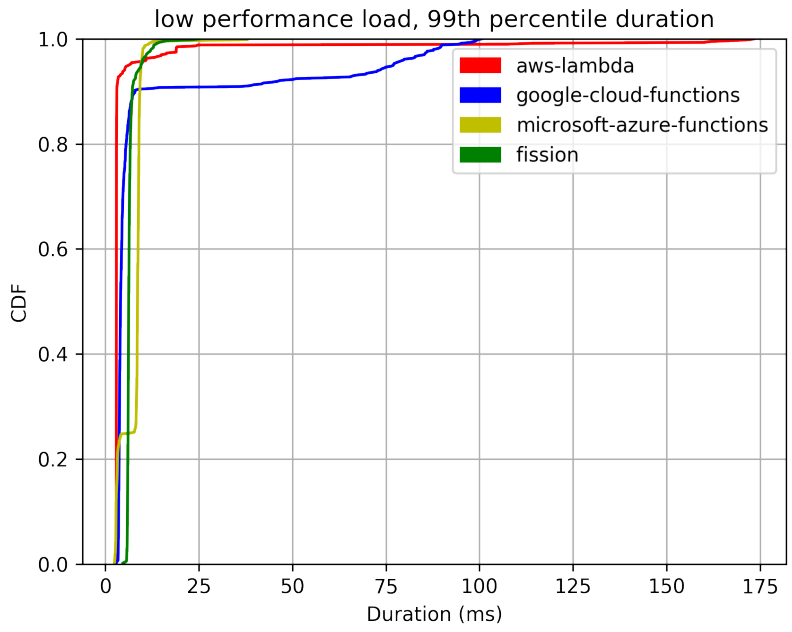

Figure 3.10: *99th percentile of low performance measurements*

Looking at the results of low performance (figure 3.10), which is the multiplication of two relatively small random matrices, we can see that almost all of the platforms performed the same way, except for Fission, which showed a little bit of faster average compute time.

Though the result times are relatively low, only 80-90 milliseconds max, the results show a relatively large deviation on all of the platforms.

Medium

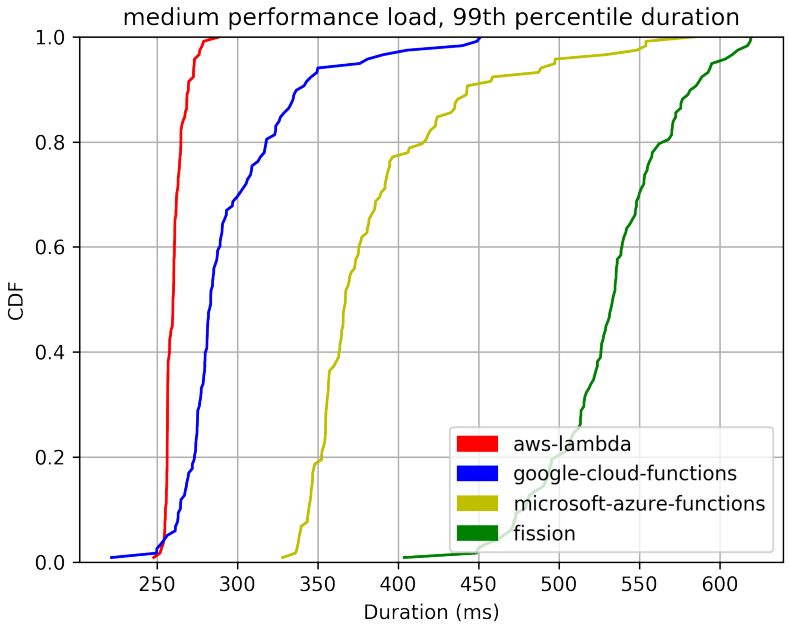

Figure 3.11: *99th percentile of medium performance measurements*

The medium complexity results show more stable performance compared to the low measurements (figure 3.11.) However, average values of the platforms varied, causing Fission to perform the worst despite being the fastest in the previous measures.

High

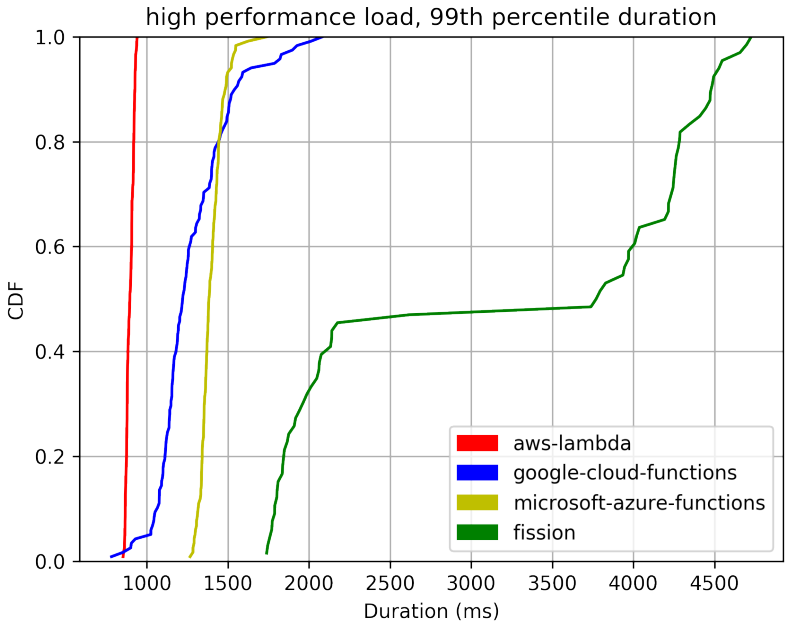

Figure 3.12: *99th percentile of high performance measurements*

The largest matrix multiplication shows somewhat similar results to the medium one. AWS Lambda performed the best here and overall too (figure 3.12.) The performance of Fission got worse compared to the other platforms.

Chapter 4

Advanced latency measurement framework

The Function as a Service space is a rapidly evolving area and because the duration of the research span for over a year, this means that since making my first measurements, a lot of things improved in these systems. As mentioned before, Google Cloud Functions came out of beta and Fission got a major update along the way as well.

Another aspect of the first results is that the measurement framework had some issues. Under bigger load, it was unable to process the incoming data fast enough thus skewing the results. Because of this I had to do measurements with lower numbers than intended.

Considering these I did some research around the current measurement tools to come up with a state-of-the-art network performance measurement system. Comparing the tools available I came to the conclusion that a tool called k6 from the Swedish company Load Impact will be the most suitable for my needs.[1] It is an open source tool written in Go and its performance was among the best compared to other measurement tools. Also its performance is matched with good usability and a variety of features. A big improvement in its architecture compared to the previous solution, is that it records the data during measurement in a NoSQL database as opposed to a file which makes the data write and read much faster removing the overhead of file writing.[17]

Another key aspect of the design of the new system was that it had to be easily portable, because the measurements are performed in different environments, so I wanted to minimize the cost of setting up the system every time when using it. K6 was a good choice in this regard as well, because it has an official Docker image, so I could configure it be used through docker-compose, which is a tool to easily manage multiple Docker containers and connect them in various ways.

To process the measurement results stored in the database, I used a command line tool called `jq`[1], which can efficiently manipulate large amounts of JSON data. With `jq`, I transformed the results to simple number lists, which could be visualized. To make these steps

[1] https://stedolan.github.io/jq/, accessed 28-October-2018

easily reproducible I created a Bash script (figure 4.1) that can be configured from the command line and with the given parameters it executes the measurements in the containers and then records the results to the file system. Then these files can be downloaded from the various environments for later use.

```
echo "Measuring $url..."

docker-compose -f ./k6/docker-compose.yml run \
  -v $PWD/tests:/tests \
  k6 run \
    -e TEST_URL=$url \
    -e CONCURRENT=$concurrent \
    -e TOTAL=$total \
    /tests/latency-test.js
```

Figure 4.1: *Part of the latency testing script of the new tool*

The last step is the CDF, cumulative distribution function, drawing from the data. For this I used Python and its data processing and visualization tools. These solutions are popular in the big data and machine learning community, so they are a great way to share our results. I used the matplotlib and scipy library to draw the CDF functions.

I aimed to create this new measurement system as general as possible, that is why I put it in a separate open source git repository to be able to handle it independently from the results. This way it can benefit others later when doing measurements of network systems.[6]

4.1 New results

I used a slightly different configuration model for the latency measurements, because the k6 framework accepts a total number of request as parameters as well (see in figure 4.2.)

Measurement types	Concurrent requests	Total number of requests
Low	10 virtual users	1000
Medium	50 virtual users	5000
High	100 virtual users	10000

Figure 4.2: *New latency measurement types*

4.1.1 AWS Lambda

Looking at the evolution of the AWS Lambda platform since the previous measurements, there have not been any major upgrade that was made officially available, other than that

new running environments are became available (both regarding Node.js and Golang). However that does not mean that there was not any improvements made to the platform.

Comparing the results with the previous measurements from Lambda we can clearly see an improvement in latency at every type of load (in figures 4.4, 4.6 and 4.8). The results became more stable and even at high load stayed under 50 ms.

4.1.2 Google Cloud Functions

As mentioned in the beginning Google Cloud Functions had a major release along the way, which means it is no longer in beta and also it is now available in multiple regions around the world. Whether this big release affected the performance of the service was not discussed publicly, but the measurements can indicate a possible improvement.

Since the previous results were measured with a Cloud Function deployed to eastern USA, because this was the only available region during the beta phase, the new measurements had to be done also in that region to get consistent results.

The results show that there was not any improvement in terms of performance, it produced the same stable results (looking at figures 4.5, 4.9). The larger values can associated with the fact that the requests travelled through the Transatlantic link.

4.1.3 Fission

Fission got a new major release as well during the research time. It changed the API in some sense, so the deployment on the Kubernetes cluster had to be changed accordingly. In the release notes they also claim performance improvements as well.

The measured results confirms this statement as well, because Fission produced almost the same latency numbers at every load type, which is among the fastest.

4.1.4 Microsoft Azure Functions

Azure Functions did not have a major release or noticeable upgrade that was documented officially, but in the first measurements it did not perform well, a lot of times produced very large numbers or the service became unavailable.

Comparing the new results we can see that the service became much more stable and faster as well. It performed similarly to the other platforms (e.g. in figure 4.7).

4.1.5 AWS Docker

The AWS Docker results serves the same role as in the previous measurements, to be a baseline for the other services.

If we look at the values from the EC2 instance and Lambda and comparing them (in figure 4.3), since they were accessed from the same source and were deployed in the same AWS region, we can see how large is the overhead of the Function as a Service abstraction layer, which consists both the container management system and the API gateway in front of it. This is a very important information for the design of a web service, because we can estimate how using a serverless technology may affect the latency of our system.

Types	EC2	Lambda	Fission	Azure Functions	Cloud Functions
Low	6 ms	23 ms	18 ms	24 ms	122 ms
Medium	8 ms	30 ms	19 ms	86 ms	126 ms
High	11 ms	44 ms	29 ms	170 ms	134 ms

Figure 4.3: *95th percentile new latency measurement results*

4.1.6 Grouped results

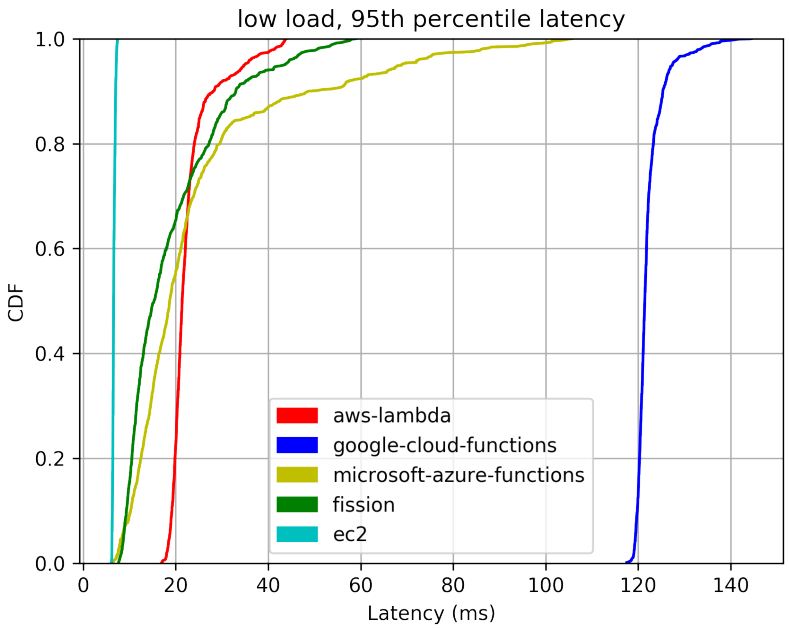

Figure 4.4: *95th percentile of second low latency measurements*

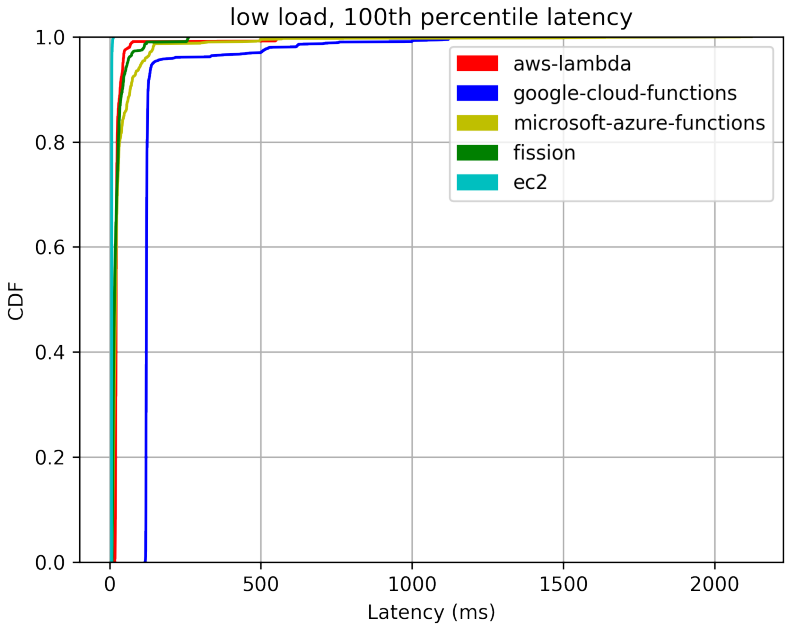

Figure 4.5: *100th percentile of second low latency measurements*

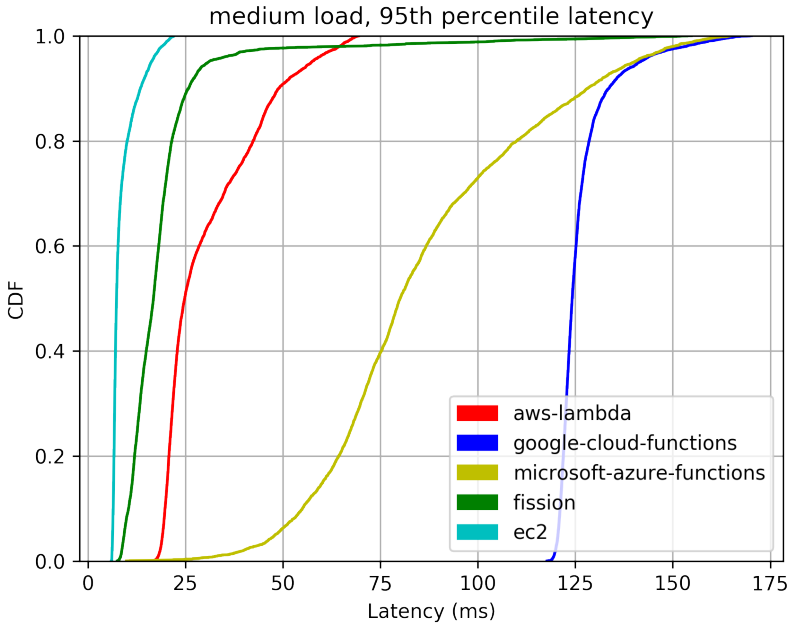

Figure 4.6: *95th percentile of second medium latency measurements*

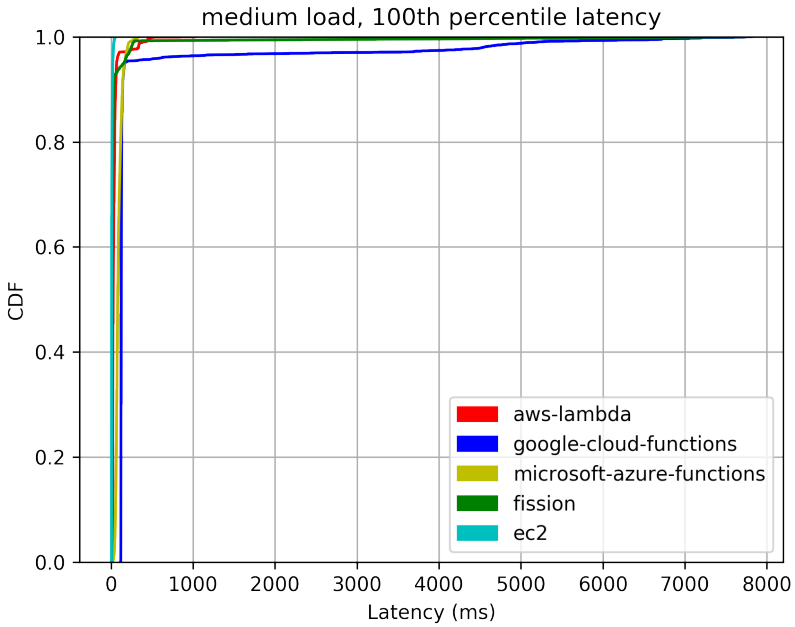

Figure 4.7: *100th percentile of second medium latency measurements*

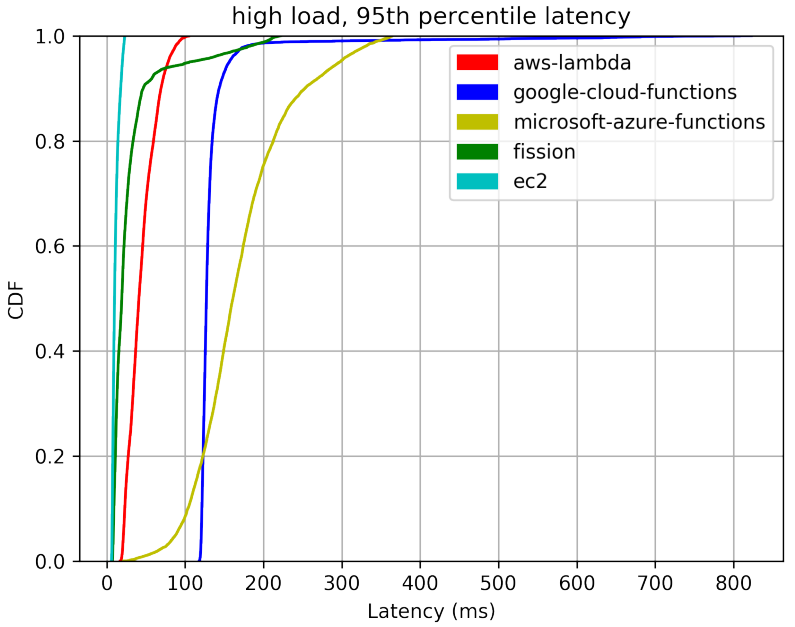

Figure 4.8: *95th percentile of second high latency measurements*

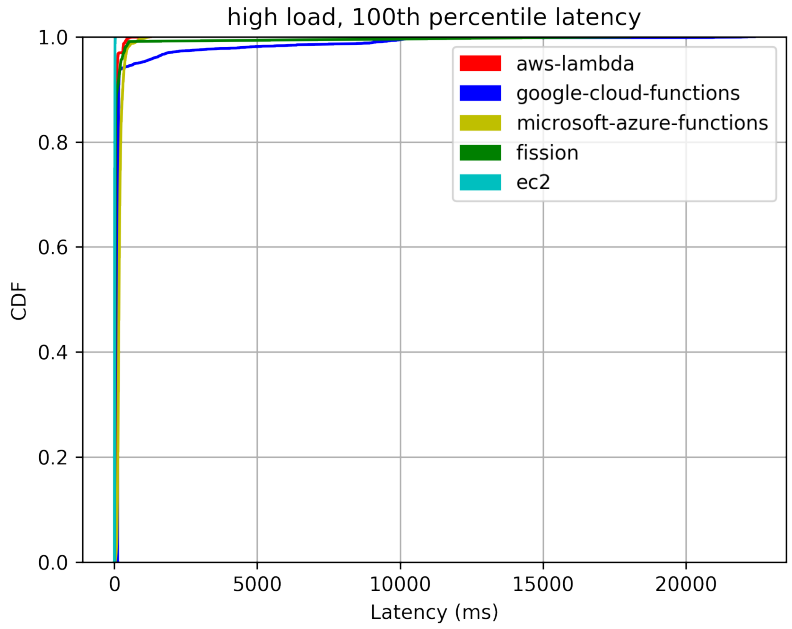

Figure 4.9: *100th percentile of second high latency measurements*

4.2 Other environments

A web service can be accessed from various sources through HTTP, so it is useful to be clear about the potential latency that we can achieve. That is why I differentiated three different environment, as mentioned earlier:

- measuring from another data center (Digital Ocean),
- measuring from a local network (figures 4.10 and 4.11),
- measuring from inside the same provider from an AWS EC2 instance to an AWS Lambda function (figure 4.12.)

The graphs above came from the Digital Ocean environment and the results from the other environments can be seen below.

Types	EC2	Lambda	Fission	Azure Functions	Cloud Functions
Low	18 ms	33 ms	42 ms	38 ms	169 ms
Medium	19 ms	38 ms	46 ms	85 ms	171 ms
High	21 ms	42 ms	88 ms	168 ms	176 ms

Figure 4.10: *95th percentile latency measurement results from local network*

Types	EC2	Lambda	Fission	Azure Functions	Cloud Functions
Low	18 ms	35 ms	47 ms	40 ms	196 ms
Medium	19 ms	41 ms	55 ms	90 ms	295 ms
High	21 ms	46 ms	114 ms	181 ms	253 ms

Figure 4.11: *100th percentile new latency measurement results from local network*

Types	95th percentile	99th percentile	100th percentile
Low	18 ms	18 ms	19 ms
Medium	18 ms	19 ms	20 ms
High	20 ms	21 ms	22 ms

Figure 4.12: *Latency results from an EC2 instance to a Lambda function*

Chapter 5

Related work

At the beginning of the research I tried to uncover all of the existing work regarding the performance measurement of serverless systems. At that time there was not so much articles about this area and most of them focused on just one particular solution.

However there are early writings mostly in the form of blog posts that perform similar measurements on multiple Function as a Service systems. Usually they do not go into the analysis of the results as deeply as in my research. There are some other aspects though that they cover that I have not gone into detail, for example the observation and comparison of cold start times, which is the latency of a request that calls a function, of which there is no associated container already running. These in the early days of serverless technologies took a significant time, but the new iterations of the existing implementations try to minimize it as much as possible, with relatively good success.[2]

Since the initial articles I studied, because of all the interest around FaaS solutions, a lot more blog posts and papers have been published and some of them are quite similar to this research. However they still usually do not cover a wide range of different solutions, especially not an open source solution to the commercial ones.

The other important information that my research uncovers and all of the other articles lack is the long term support, the evolution and the improvements over time for a serverless technology.

Besides the articles about the various performance metrics, the ones that were more helpful for my research were the posts describing the best practices and the tools needed for a convenient development and deploy flow.[17]

There are a lot of tools available for serverless function management, the most widely used is the Serverless Framework that I used, but recently AWS released its own similar tool exclusively for Lambda.[20]

Only just by looking at how many things evolved and improved during the research in the field of Function as a Service technologies, we can conclude that it has a very important role

in today's and the future of cloud architectures. This model definitely has the potential to by the next standard in web service development, because of its cost effective only pay for the computational time financial model, its low operational cost and easy scalability. But it is important to design these services carefully, because a bad direction in development could mean the end of this idea. It is also important for developers and companies to know when they can safely switch over to these new technologies and for which use cases they are most suitable.[4]

That is why the measurements and comparisons like my research, are a key information source that can help developers and companies to make a good decision in a design of a cloud system.

Chapter 6

Conclusion

In this article we looked at the current state of the Function as a Service platforms and their capabilities. The main reason for these measurements is to determine whether this technology can be used today in real world applications.

As a result of this research, numerous measurement information and numbers became available that can be used later to identify other behaviours and make observations besides the ones I made. Also the other important output of the work is the public and open source performance testing project that provides a robust and automated solution to perform any kind of load tests with web services.

All of the observed platforms have pros and cons, but overall it can be stated that FaaS solutions are viable for many use-cases today. If our application is not latency critical and it is important for the tools to be as simple as possible, which covers a wide range of the Internet today, provider hosted FaaS platforms, for example AWS Lambda, are a feasible option to go with. If we have more resources, put into more developer effort and we want better latency, going with a self hosted, open source FaaS framework, such as Fission, is a clear possibility.

The underlying technology of FaaS is based on containers and we looked at the computational performance of these ad-hoc created instances, also how they scale and it is safe to say that for most use-cases their performance will not be a bottleneck, but because of the limited hardware capabilities of the available instance types, currently it might not be most suitable for large and computational heavy tasks.

Another important aspect is that these technologies are considerably new and evolving. The beta phase of Google Cloud Functions and the early hiccups of the Microsoft Azure Functions are very much showed this, but as we observed later these problems can be dealt with and companies are working hard to get rid of them. That is why when choosing a service it is important to keep in mind that these are novel solutions and this sometimes can affect the usability and performance in a negative way.

However the latest measurements and observations show that the platforms are improving very fast and they cover more and more use-cases. Also currently almost every technology

got to the point where it can be safely used in production, so if we do not want worry about operational management and scaling in our service, Function as a Service solutions are a great way to achieve our goals.

Acknowledgement

I would like to thank dr. Maliosz Markosz my consultant, who greatly helped my research with pointing me in the right directions, giving good advice and supportive comments.

I would like to also thank dr. Simon Csaba, who supported my work with very helpful instructions and recommendations.

The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013).

Bibliography

[1] Load Impact AB. K6. https://k6.io, 2017. [Online; accessed 28-October-2018].

[2] Matt Billock. The Serverless Performance Shootout. https://dzone.com/articles/the-serverless-performance-shootout, 2017. [Online; accessed 28-October-2018].

[3] Microsoft Corporation. Microsoft Azure Functions. https://azure.microsoft.com/services/functions, 2018. [Online; accessed 28-October-2018].

[4] I. Baldini et al. Serverless computing: Current trends and open problems. *Chaudhary S., Somani G., Buyya R. (eds) Research Advances in Cloud Computing (pp. 1-20)*, 2017.

[5] P. R. Brenner G. McGrath. Serverless computing: Design, implementation, and performance. *37thIEEE International Conference on Distributed Computing Systems Workshops*, 2017.

[6] Gellert Hegyi. Performance testing of Serverless platforms. https://github.com/gerhardberger/performance-testing, 2018. [Online; accessed 28-October-2018].

[7] IBM. IBM OpenWhisk. https://www.ibm.com/cloud/functions, 2018. [Online; accessed 28-October-2018].

[8] Amazon.com Inc. AWS Lambda. https://aws.amazon.com/lambda, 2018. [Online; accessed 28-October-2018].

[9] Galactic Fog IP Inc. Gestalt. http://www.galacticfog.com, 2018. [Online; accessed 28-October-2018].

[10] Platform9 Systems Inc. Fission. https://fission.io, 2018. [Online; accessed 28-October-2018].

[11] Serverless Inc. Serverless Framework. https://serverless.com, 2018. [Online; accessed 28-October-2018].

[12] Badri Janakiraman. Serverless. https://martinfowler.com/bliki/Serverless.html, 2016. [Online; accessed 28-October-2018].

[13] Kubeless. Kubeless. https://kubeless.io, 2018. [Online; accessed 28-October-2018].

[14] Niko Köbler. Serverless Compute Manifesto. https://www.n-k.de/2016/12/serverless-compute-manifesto.html, 2016. [Online; accessed 28-October-2018].

[15] Google LLC. Google Cloud Functions. https://cloud.google.com/functions, 2018. [Online; accessed 28-October-2018].

[16] Shoreditch Ops Ltd. Artillery. https://artillery.io, 2018. [Online; accessed 28-October-2018].

[17] Ragnar Lönn. Open Source Load Testing Tool Benchmarks V2. https://blog.loadimpact.com/open-source-load-testing-tool-benchmarks-v2, 2017. [Online; accessed 28-October-2018].

[18] Juan Anibal Micheli. Awesome Serverless. https://github.com/anaibol/awesome-serverless, 2018. [Online; accessed 28-October-2018].

[19] OpenFaaS. OpenFaaS. https://github.com/openfaas/faas, 2018. [Online; accessed 28-October-2018].

[20] Pedro Fernando Marquez Soto. Running Go AWS Lambdas locally with SLS framework and SAM. https://medium.com/a-man-with-no-server/running-go-aws-lambdas-locally-with-sls-framework-and-sam-af3d648d49cb, 2018. [Online; accessed 28-October-2018].

[21] Josef Spillner. Snafu: Function-as-a-service (faas) runtime design and implementation. *arXiv preprint arXiv:1703.07562*, 2017.