A Benchmark Suite for Serverless Computing

AWS Lambda - Microsoft Azure Functions - Google Cloud Functions - IBM Cloud Functions

February 2020

Master thesis of Pascal Maissen
Staldenstrasse 58, 3920 Zermatt
pascal.maissen@unifr.ch
Student-Number: 13-208-335
University of Fribourg

Supervised by:

Prof. Dr. Pascal Felber
Dr. Valerio Schiavoni

Acknowledgements

I would first like to thank my thesis advisers Prof. Dr. Pascal Felber and Dr. Valerio Schiavoni of the university of Neuchâtel. They both allowed me to work on this topic by myself but were always available when I needed help or had questions and gave me valuable comments on this thesis.

I would also like to acknowledge my girlfriend Stephanie Abgottspon and my sister Samina Maissen for proof-reading this thesis.

Finally I need to express great gratitude to my parents and my grandfather for supporting me upon this point in my life. Not only did they support me financially but they also encouraged me to pursue my dream of becoming a graduate of the university in computer science. This accomplishment would not have been possible without them.

Thank you

Pascal Maissen

Abstract

Currently there is a big trend towards cloud computing. Among all the services the cloud offers, serverless computing is very popular and every large cloud service provider offers a serverless platform. However, there has not been much effort to test, benchmark or compare these services. This thesis tries to take an approach on benchmarking serverless computing with building a suite that everyone can use.
This suite provides five different tests to benchmark on four different clouds, highly automated deployment and cleanup of these tests, a testing utility for comparing cloud providers, a heavy benchmark to load test the serverless platforms and a pricing calculator to estimate hypothetical and actual costs. Everything is packaged with Docker and is easy to use.
This benchmark suite is completely open source and the code and the documentation can be found at:

`https://github.com/bschitter/benchmark-suite-serverless-computing`

Keywords: Serverless, Serverless Computing, Benchmarking, AWS Lambda, Microsoft Azure Functions, Google Cloud Functions, IBM Cloud Functions, Function as a Service (FaaS)

Table of Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Cloud Computing . . . 2
1.3 Serverless Computing . . . 4
1.4 Benchmarking . . . 5

2 Benchmarking Serverless Computing **6**
2.1 Choosing Serverless Computing Providers . . . 6
2.1.1 Amazon Web Services Lambda . . . 7
2.1.2 Microsoft Azure Functions . . . 8
2.1.3 Google Cloud Functions . . . 8
2.1.4 IBM Cloud Functions . . . 9
2.2 Choosing Runtimes . . . 9
2.2.1 Node.js . . . 10
2.2.2 Python . . . 11
2.2.3 Go . . . 11
2.2.4 .NET Core . . . 11
2.3 Test Functions . . . 12
2.3.1 Latency Test . . . 12
2.3.2 CPU Test (Factorization) . . . 13
2.3.3 CPU Test (Matrix Multiplication) . . . 14
2.3.4 I/O Test . . . 14
2.3.5 Custom Test . . . 15

3 Implementation **16**
3.1 Overview . . . 16
3.2 Requirements . . . 18
3.3 Deployment and Cleanup . . . 19
3.4 Testing . . . 20
3.5 Benchmarking . . . 20

3.6 Pricing 21

4 Results **22**
4.1 Tests 22
4.1.1 Latency Test 22
4.1.2 Cold Start Test 23
4.1.3 General Test 25
4.1.4 Load Test 27
4.2 Pricing 30
4.3 Evaluation of all services 32
4.4 Advantages and disadvantages of serverless computing 35

5 Conclusion **37**
5.1 Summary 37
5.2 Future Work 37

Appendix A Flow charts **38**

Appendix B Results **43**

Appendix C Miscellaneous **52**

List of Figures **54**

List of Tables **55**

Glossary **56**

References **57**

Chapter 1

Introduction

This chapter explains the motivation behind the project and the importance to test and benchmark the different available platforms from cloud providers. It introduces the topics of cloud computing and particularly serverless computing and briefly explains the term benchmarking.

1.1 Motivation

In today's world cloud computing is for many companies and organizations a good and maybe the best option to set up their computing infrastructure or migrate to it. Smaller or newer companies often cannot afford to invest in high-performance hardware which they also need to manage themselves. Sometimes companies don't have the knowledge to maintain hardware properly and just want their databases and web servers to work, instead of worrying about a hardware failure or a power outage. Furthermore most businesses want to focus on their core business which many times means using computing resources and tools instead of managing them.
This thesis is going to investigate a specific region of cloud computing; generally known as *serverless computing* or also often referred to as Function as a Service (FaaS) (although this is only a subset of serverless computing). As the name suggests, it is a serverless service meaning the user does not have to set up a server or manage its operating system. In general, when using serverless computing, the user provides the application code, deploys it to the cloud and the cloud provider handles the load the application generates and other important elements like availability and scaling. Hence, the name function is often used.
However, some issues remain unresolved and some questions stay open. Which cloud provider to choose to run an application on? Can the cloud provider handle the load? How much is it going to cost? This thesis tries to answer these questions by providing a benchmark and testing suite for serverless computing. With this suite a potential cloud user can run some tests to see how each different cloud provider performs and the suite helps to make the decision easier.
So far, only few efforts have been made to test or benchmark serverless computing [FGZ+18, KNJ18, LSF18, MFGZ18] and related research has only given little attention to serverless computing [GZC+19]. There is some more research which treats the topic serverless computing but not in regards to benchmarking [BCC+17, CIMS19, EIST17].

1.2 Cloud Computing

The National Institute of Standards and Technology (NIST) defines cloud computing as a model which enables ubiquitous, on-demand network access to computing resources which can be rapidly and easily provisioned and released [MG11]. Computing resources can be servers (dedicated hardware or VMs), storage, applications, services, etc. Those resources are managed by a cloud service provider in their data centers and are normally accessible to everyone on a pay per usage model. The NIST also states the following **essential characteristics** of cloud computing [MG11].

- **On-demand self service:** A user can provision computing resources automatically and by himself.
- **Broad network access:** The resources are available over network, typically over a website or a Command Line Interface (CLI).
- **Resource pooling:** Physical and virtual resources are pooled, used in a multi-tenant model and dynamically assigned depending on demand. The user has no control neither knowledge of the specific location where his resources are allocated, only on a higher level e.g. which data center.
- **Rapid elasticity:** Services and resources can be elastically and mostly automatically provisioned and scale rapidly. To a user, the available resources seem unlimited at any time.
- **Measured service:** Resource allocation happens automatically and is optimized by gathering metrics about the resources. The usage of resources can be controlled and monitored transparently for both parties and also benefits both.

Besides the essential characteristics there are also three service models and four deployment models which will be described in the following [MG11, IBM19d].

Service models

- **Infrastructure as a Service (IaaS):** The consumer can provision processing (i.e. servers), storage and network components. The user does not control the underlying infrastructure. However, he has control over the operating system, storage and applications. There is also only limited control over the network (i.e. firewalls). A good example would be a VM on AWS Elastic Compute Cloud (EC2).
- **Platform as a Service (PaaS):** The user can deploy applications on the cloud infrastructure within the provider's supported programming languages, services and tools. He has no control over the OS nor the storage, only the application itself, its data and some configuration parameters. An example for PaaS is Google App Engine.
- **Software as a Service (SaaS):** The consumer uses the provider's applications as they are. The user has no control over the applications capabilities. Usually such software is accessed through a web browser (i.e. website) or other clients. An example of this model is Microsoft Office 365.

Figure 1.1 shows where the responsibilities are with which service model.

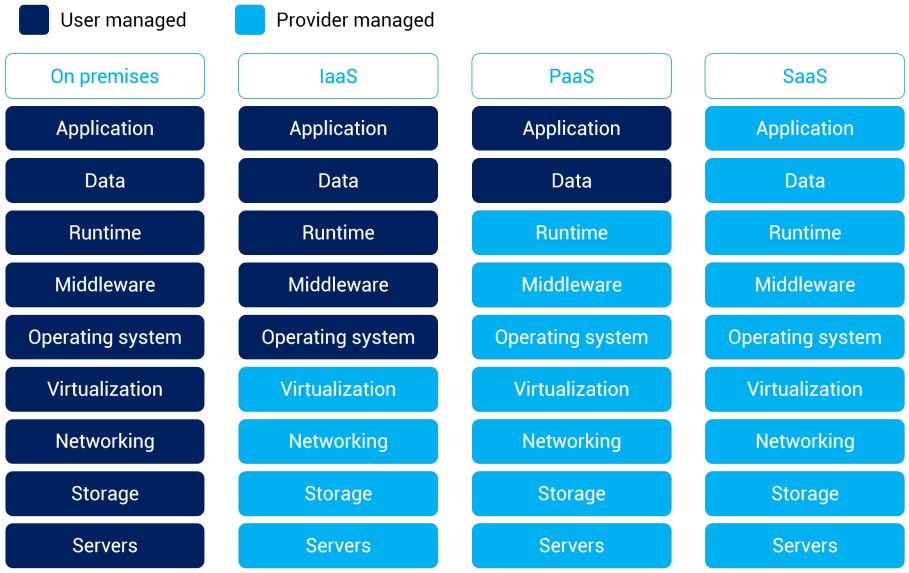

Figure 1.1: On premises - IaaS - PaaS - SaaS
Source: Alibaba Cloud [Ali19]

Deployment models [MG11]

- **Private cloud:** The infrastructure is exclusive to an organization. It can be owned and managed by the organization itself, but also by a third party. The private cloud can exist on or off premise.
- **Community cloud:** A community cloud is very similar to a private cloud with the difference that it is provided for a specific community with common interests. It can be owned and managed by the community or a third party and can also be on or off premise.
- **Public cloud:** The infrastructure is generally available to the public. It is typically owned and managed by the organization that runs it and is located on their premises.
- **Hybrid cloud:** A hybrid cloud is a mixture of the above three types. Each type remains a separate unit, but is interconnected with the other units. In practice, this is often seen with companies who switch to a public cloud. They use a combination of private and public cloud and can therefore migrate application by application to the public cloud until all runs in the public cloud.

The German Federal Office for Information Security also shares this definition of cloud computing from the NIST and has adopted it in principle more or less literally [Bun19].
This thesis will be only treating public clouds and a slightly different service model called Function as a Service (FaaS). It will be explained in the next section 1.3.

1.3 Serverless Computing

The term *serverless* reckons by no means that there are no servers involved, but rather that the user does not have to manage them, because the cloud provider does. IBM's definition of serverless computing is: "Serverless is an approach to computing that offloads responsibility for common infrastructure management tasks (e.g., scaling, scheduling, patching, provisioning, etc.) to cloud providers and tools, allowing engineers to focus their time and effort on the business logic specific to their applications or process." [IBM19e]
Using serverless technologies requires much less expertise than non serverless self managed implementations. Although those technologies might come with certain limitations or performance bottlenecks that won't fit to the user's needs.
The most important key features of serverless computing are the following: No server or infrastructure management of the user is required, the workload is scaled dynamically and automatically and it is usually paid per usage, e.g. only charged for the occupied storage in a service [AWS19, Mic19b].
Mainly three popular serverless categories exist, namely FaaS (Function as a Service, e.g. AWS Lambda), DBaaS (Database as a Service, e.g. Microsoft Azure Database for PostgreSQL) and STaaS (Storage as a service, e.g. Google Cloud Storage [Goo19]). This thesis will treat the domain of FaaS.

FaaS (Function as a Service)

FaaS is defined in between the PaaS and the SaaS service model, since the data is managed by the provider but not the application (see figure 1.1). With most cloud service providers, the user can upload an application (this can be in the form of source code, binaries or even a Docker image) and define a trigger (HTTP, Storage, Database, etc.) which invokes the function. This can be useful for application backends and data processing [AWSa, Gooa]. By the help of this technology, users can implement their services without either buying or renting a server or a VM in the cloud. They can for example create a backend service without thinking about hardware and server management. Figure 1.2 illustrates a typical FaaS infrastructure.

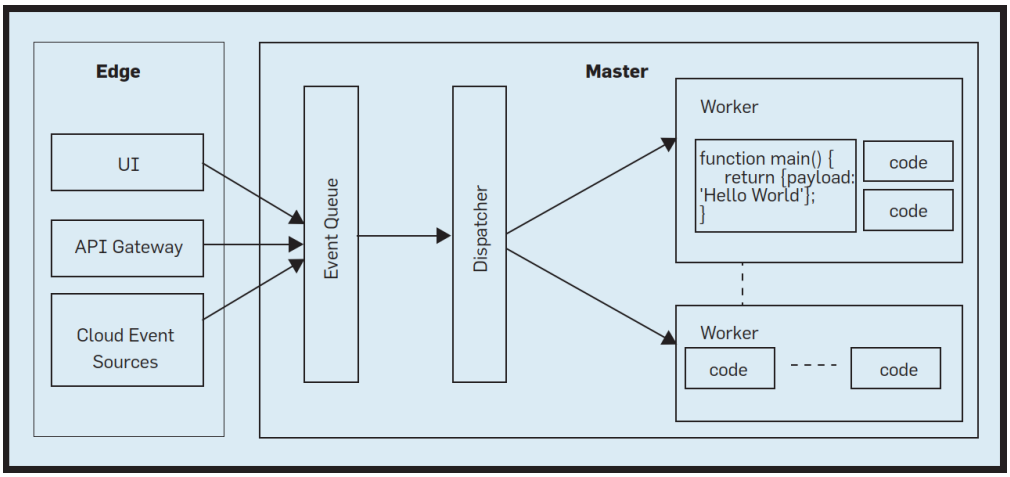

Figure 1.2: High-level serverless FaaS platform architecture. Source: [CIMS19]

1.4 Benchmarking

Benchmarking is a method to analyze and test the performance of a system, to discover its benefits and weaknesses and to compare it directly to other systems. In this thesis, the systems will be serverless platforms and they will be tested with different applications to see how fast they run, how well they scale and how much they cost. In order to get meaningful results, these tests should be executed as similar as possible on each system and repeated enough times to avoid coincidental data. In the following chapters the systems, the tests, the benchmark process and the results will be carefully discussed and explained in detail.

Chapter 2

Benchmarking Serverless Computing

In this chapter, the choice of serverless computing providers and the choice of runtimes respectively programming languages is derived. Furthermore, the implemented tests will be explained in detail and it is shown what their effect and importance is.

2.1 Choosing Serverless Computing Providers

There are a lot of cloud providers available and many of them are trying to grow and therefore investing more and more in their infrastructure. So this benchmark suite could have taken into account as many cloud providers as possible, assumed they provide serverless computing, but that would have exceeded the scope of this thesis. Figure 2.1 illustrates on the left hand side the global revenue of the cloud infrastructure services market which has reached nearly $23 billion in the second quarter of 2019. What's more interesting for choosing cloud providers for this benchmark suite is the market share. As one can see in figure 2.1 on the right hand side the market share of cloud computing is mainly dominated by a few big players: Amazon, Microsoft, Google, IBM and Alibaba. Amazon still is the unprecedented leader in this field of business. Nevertheless, other competitors are heavily investing in the cloud business. Google will invest 3 billion euros in its European data centers as Sundar Pichai, the CEO of Google, stated in the Google blog [Pic19]. And also Microsoft has recently been given a $10 billion contract by the US Department of Defense to transform the military's cloud computing systems [CSS19, US 19].
For the above mentioned reasons, the following top four providers were taken into consideration.

- Amazon Web Services
- Microsoft Azure
- Google Cloud
- IBM Cloud

All of them offer a serverless platform which will be described for each cloud provider in the following sections.

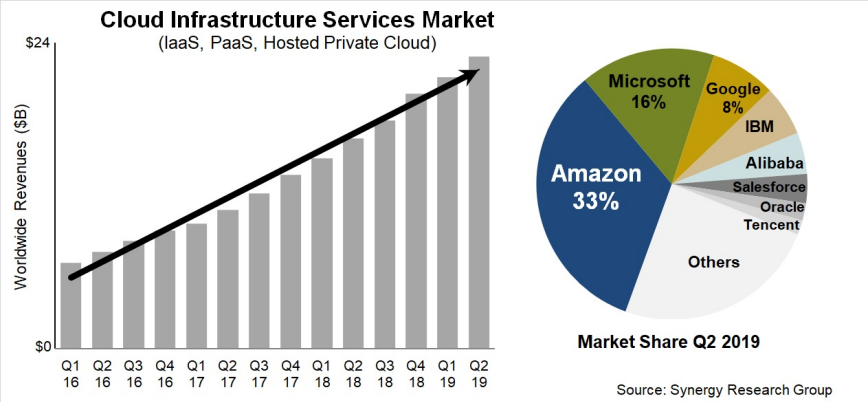

Figure 2.1: Cloud Infrastructure Services Market Share
Source: Synergy Research Group [Syn19]

Remark. At the beginning of this thesis, a deployment on Apache OpenWhisk was considered on the private cluster of the university of Neuchâtel. After several time wasting and failed attempts to set up OpenWhisk and contradicting documentation on the OpenWhisk website respectively the GitHub page of OpenWhisk the idea was dropped. Since IBM uses an implementation based on OpenWhisk, the performance is expected to be similar. Nevertheless, it would have been interesting to compare those two very similar or even identical systems on a private and on a public cloud.

2.1.1 Amazon Web Services Lambda

The service of Amazon is named *AWS Lambda* [AWSa]. It was released on the 13th of November in 2014 [AWSe] and today it supports six different runtimes including seven different programming languages [AWSf]. Amazon claims, depending on the region where the function is deployed, a concurrency of 500 to 3000 instances will be provisioned at most [AWSc]. The memory allocated to a function instance can vary from 128 MB up to 3008 MB in steps of 64 MB [AWSb]. AWS Lambda is available in 18 out of 18 publicly accessible regions (not counting two chinese regions) [AWSg]. Amazon also mentions that the CPU power will increase linearly with memory allocation and at 1792 MB the function will get 1 vCPU [AWSb]. As the paper by Wang et al. [WLZ+18] suggests, AWS Lambda uses mostly two different CPUs, the Intel Xeon E5-2666 and the Intel Xeon E5-2680, which have a clock rate of 2.90 GHz and 2.80 GHz respectively. This information was extracted by Wang et al. from the file `/proc/cpuinfo` in the Linux operating system.

Remark. From my own experience I can tell that cloud providers generally don't like to tell their customers every detail of hardware they use or how exactly services are implemented. The first point might be true because they don't want to tell a customer what exact CPUs he gets, because then he will certainly complain if it differs from the specification and the provider has to take responsibility. The second point should be obvious for economic and competition related reasons.

Knowing that, one can more or less estimate the theoretical computing power that the virtual CPU will provide.

Remark. The pricing of AWS Lambda and all the other services will be discussed in section 4.2 Pricing.

2.1.2 Microsoft Azure Functions

Azure offers the service called *Azure Functions* [Mica]. It was first released in March 2016 in preview (Microsoft's term for beta) and then later in November 2016 it became generally available [Kir]. Currently there are five supported runtimes including seven different programming languages [Micg]. Compared to the other three cloud providers, Azure offers three different hosting plans [Micd]:

> **Consumption plan:** It adds and removes instances dynamically depending on the load on the function and cost only arise when functions are running. This is the most 'serverless' option among those three.
>
> **Premium plan:** The premium plan is similar to the consumption plan but offers more integration and control over the functions. Instance sizes can be chosen and instances can be pre-warmed. The cost is calculated with CPU and GB memory used per second.
>
> **App Service plan:** How many and on which VMs the functions run can be decided in the App Service plan. Scaling happens manually, time based or based on metrics such as CPU usage.

This thesis will only consider the consumption plan, as it is the default plan and fully managed, and therefore *more* serverless than the others. Additionally, it is similar to the services of the other providers.
There are currently three different generations of the service available [Micc], this project uses generation 2.
Azure Functions can deploy up to 200 instances and provide up to a maximum amount of 1.5GB memory [Micd]. The service can run either on Windows or Linux and is offered in 28 out of 46 publicly accessible regions [Micf], but the consumption plan is only available in 11 regions for both Linux and Windows. For computing power, Azure has its own term named Azure Compute Unit (ACU). The instances in the Azure functions consumption plan have an ACU of 100 which is about the equivalent of 1 vCPU. Azure functions is likely to use the VM type *Av2* which is the only one who corresponds to the declared ACU [Mice]. These VMs use three different CPUs: the Intel Xeon 8171M at 2.1 GHz, the Intel Xeon E5-2673 v4 at 2.3 GHz and the Intel Xeon E5-2673 v3 at 2.4 GHz [Mice].

2.1.3 Google Cloud Functions

On the Google Cloud Platform, the serverless service is simply called *Functions* [Gooa]. The service was first released on the 9th of March in 2017 in beta and then on the 24th of July in 2018 as stable and usable in production [Gooe]. Compared to AWS and Azure that is two and a half years respectively one year later for the first release. Google Cloud Functions currently only supports three programming languages [Goof]. The documentation does not mention a limit of maximum allocated instances per function. However, it states that at maximum 1000 functions can be concurrently in execution [Good]. Possible options for CPU and memory allocation per instance are shown in table 2.1 [Gooc]. The service is available in seven out of twenty regions [Goog].

Memory	128MB	256MB	512MB	1024MB	2048MB
CPU	200MHz	400MHz	800MHz	1.4 GHz	2.4 GHz

Table 2.1: Google Cloud Functions - Possible memory allocation and corresponding CPU frequency
Source: Google Cloud documentation [Gooc]

Google does not mention what type of CPU they use for functions on the underlying infrastructure. The CPU type could also not be extracted by Wang et al. [WLZ+18] and the file `/proc/cpuinfo` does not show information on the CPU model name. It shows however information for the fields `vendor_id`, `cpu_family` and `model`. Those values reveal that Intel processors are used and can give an indication from which generation the CPU is. See the example file content C.1 in the appendix.

2.1.4 IBM Cloud Functions

The solution of IBM is called Cloud Functions [IBMb]. It is based on Apache OpenWhisk which is an open source serverless cloud platform using Docker containers [Apa]. IBM Cloud Functions supports nine different runtimes including a Docker runtime [IBM19a]. The service is restricted to 1000 concurrently active executions (also counting queued for execution) per namespace [IBM19c]. However this limit can be increased for a specific business case but needs to be applied for via the ticketing system of the IBM support [IBM19c]. Memory allocation can be set from 128 MB to 2048 MB in steps of 32 MB. IBM Cloud Functions is available in five out of six regions [IBM19b], but one region has no support for cloud foundry [IBMa] and can therefore not be deployed with the CLI and is not included in this project. The documentation does not mention anything on CPU or machines they are using. The file `/proc/cpuinfo` reveals that one possible CPU processing the functions is the Intel Xeon E5-2683 v3 at 2.00 GHz. See the example file content C.2 in the appendix.

2.2 Choosing Runtimes

After the technical description and specification of each provider, the following section will deduct which runtimes respectively programming languages were chosen and why.
Table 2.2 shows an overview of all supported runtimes for each cloud service provider mentioned in section 2.1. On the right hand side there is the sum for each runtime, which indicates the number of cloud service providers supporting that runtime.
It is obviously more interesting to compare runtimes and programming languages that are supported in multiple clouds and are therefore directly comparable. As the table 2.2 shows Node.js is supported in each cloud, Python in all except Azure with Windows as OS, .NET Core in all but Google and so forth. For this thesis, the top four runtimes where chosen to limit the scope. A brief summary of each runtime will be given in the following subsections.

Remark. Azure has currently 3 generations of its function service available. generation 1 is currently in maintenance mode and generation 2 and 3 are generally available. The available runtimes in the next section

	AWS	Azure		Google	IBM	Sum
		Linux	Windows			
Node.js	yes	yes	yes	yes	yes	4.0
Python	yes	yes	no	yes	yes	3.5
.NET Core	yes	yes	yes	no	yes	3.0
Go	yes	no	no	yes	yes	3.0
Java	yes	no	yes	no	yes	2.5
Ruby	yes	no	no	no	yes	2.0
Swift	no	no	no	no	yes	1.0
PHP	no	no	no	no	yes	1.0
Docker	ECS	yes	no	Cloud Run	yes	1.5

Table 2.2: Supported runtimes in serverless computing. Data source: [AWSf, Micg, Goof, IBM19a]
Remark: Since Azure has Linux and Windows as underlying OS, only half a point is counted per OS.

will only consider generation 2 (which is also used in this project):

2.2.1 Node.js

Node.js is a JavaScript runtime and is built on Chrome's V8 JavaScript engine [Noda]. As the name indicates, JavaScript is a scripting language and was first released on the 4th December 1995 by Netscape [Wik19]. It was mostly used in addition to HTML and CSS in web browsers [Wik19]. Because of the Node.js framework and the Node Package Manager (NPM), JavaScript is nowadays a popular language to implement all kinds of applications. Table 2.3 shows supported Node.js versions of each cloud provider.

Node.js	AWS	Azure	Google	IBM
6.x	no	no	yes	no
8.x	no	yes	yes	yes
10.x	yes	yes	yes	yes
12.x	yes	no	no	no

Table 2.3: Supported Node.js runtimes. Data source: [AWSf, Micg, Goof, IBM19a]
Remark: The yellow color means *deprecated* and *beta* for version 6.x respectively 10.x.

This benchmark suite will deploy all Node.js applications in version 10 as all providers support this version. In particular, AWS uses version 10.x [AWSf], Azure does not specify more than version 10 [Micg], Google uses version 10.15.3 [Goob] and IBM implements version 10.15.0 [IBM19a].

Remark. Google has Node.js version 10 still in beta while Node.js version 8 End-of-life was on December 31, 2019 [Nodb].

2.2.2 Python

Python is a universal, open-source and very popular programming language. It was released in 1991 and created by Guido van Rossum [w3s] and runs basically anywhere [Pyta]. Python was designed to be very easy to learn and to have readable code [w3s]. It only uses indentation and whitespaces to define the scope of loops, functions and classes [w3s]. Because of those characteristics, developers can implement new features very fast. Cuong Do, software architect of YouTube, said: "Python is fast enough for our site and allows us to produce maintainable features in record times, with a minimum of developers." [Pytb]. Table 2.4 shows which of the four cloud provider supports which Python versions.

Python	AWS	Azure	Google	IBM
2.7	yes	no	no	yes
3.6	yes	yes	no	yes
3.7	yes	yes	yes	yes
3.8	yes	no	no	no

Table 2.4: Supported Python runtimes. Data source: [AWSf, Micg, Goof, IBM19a]

All clouds support version 3.7, therefore this version will be used in the benchmark functions.

2.2.3 Go

Go is a procedural open source programming language developed by a team at Google and other contributors [Go19a, Go19b]. It is a relatively new language and was first released in March 2012 [Go19b]. It is compiled and therefore more efficient than interpreted languages and it has good concurrency mechanisms to benefit from today's multi-core architecture [Go19a]. Go also has a garbage collector in contrast to C [Go19a]. On the Go website the following statement can be found: "It's a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language." [Go19a]. Today, Go is a popular language given that is is easy to learn and write like Python but also very efficient like C. Table 2.5 shows the supported Go versions.

Go	AWS	Azure	Google	IBM
1.11	yes	no	yes	yes

Table 2.5: Supported Go runtimes. Data source: [AWSf, Micg, Goof, IBM19a]
Remark: AWS supports all versions of Go 1.x, depending on the version the binary was compiled with and deployed to Lambda.

This thesis will use Go 1.11 for its benchmark functions.

2.2.4 .NET Core

.NET Core is a free and open-source software framework developed by Microsoft and its community [Mic19a]. The first version was released in June 2016 [Mic16]. It supports as languages C#, F# and Visual Basic [Mich] and is currently on version 3.1 which was released in December 2019 [Lan19]. Microsoft has declared its love

for Linux when Satya Nadella in the end of 2014 at a Microsoft Cloud Briefing in San Francisco presented a slide which stated "Microsoft ♥ Linux" [Tea15]. Since then the company has embraced Linux, open-source software and cross-platform support. Table 2.6 shows supported .NET Core versions.

.NET Core	AWS	Azure	Google	IBM
2.1	yes	no	no	no
2.2	no	yes	no	yes
3.1	no	no	no	no

Table 2.6: Supported .NET Core runtimes. Data source: [AWSf, Micg, Goof, IBM19a]

Because there is no common supported version, this project will use 2.1 on AWS and 2.2 on Azure and IBM. The language the test functions are implemented is C#.

2.3 Test Functions

This section will briefly explain all the implemented test functions, what they test and how they are implemented. All the functions are implemented with a HTTP trigger, since it is one that all clouds support and is the easiest to test across all platforms. For the same programming language and the same test, the implementation will vary from cloud to cloud a little bit. This is due to slightly different function calls and return statements each cloud defines on their own. However, these differences are not relevant regarding performance results.

2.3.1 Latency Test

The latency test is fairly simple. The function is called and then immediately returns a small JSON body with HTTP status code 200. The test function is intended to be as fast and simple as possible to measure latency for each cloud and runtime. The following code listing 2.1 shows exemplary the implementation of the latency test on AWS in Node.js.

Listing 2.1: Latency test implementation on AWS in Node.js

```
exports.handler = function(event, context, callback) {
    const res = {
        statusCode: 200,
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({
            success: true,
            payload: {
                'test': 'latency test',
            }
        })
    };
  callback(null, res);
}
```

2.3.2 CPU Test (Factorization)

This test describes the first of the two CPU tests. The CPU is the component of a system which does effectively do the calculations of a program. It is therefore the most important criterion of serverless computing and logically also the most expensive. This test is targeting mostly the CPU by calculating all integer factors/divisors imperatively of an integer number. Listing 2.2 shows pseudo code of such a number factorization.

Listing 2.2: Factorization test pseudo code

```
for(i = 1; i < SquareRoot(Number), i++) {
    if(Number modulo i == 0) {
        factors.add(i)
        if(Number / i != i) {
            factors.add(Number / i)
        }
    }
}
```

The algorithm works as follows: one iterates from 1 to the square root of the number N to be factorized. For each value i it is tested if N is dividable by i without rest (modulo operator). If so, a factor of N has been found and i is added to the results. In addition, if N divided by i does not equal i itself, the matching part x of i has also been found where $x \cdot i = N$. This is also the reason why only the numbers up to the square root of N need to be considered. The algorithm has a complexity of $O(n^{\frac{1}{2}})$.

Remark. The algorithm is not to be confused with prime factorization. In this case, all factors of a number are calculated but with prime factorization only prime numbers are calculated. The approach and results share some characteristics.

2.3.3 CPU Test (Matrix Multiplication)

The matrix multiplication is the second CPU test in this benchmark suite. The user can input a number n which defines the width and height of two matrices. The two matrices are both filled with random integer numbers between 0 and 100. The product of those two matrices is calculated by multiplication. The algorithm is defined in listing 2.3 in pseudo code.

Listing 2.3: Matrix multiplication test pseudo code

```
matrixA = randomMatrix[n][n]
matrixB = randomMatrix[n][n]
matrixMult = [n][n];

for(i = 0; i < matrixA.height; i++) {
    for(j = 0; j < matrixB.width; j++) {
        sum = 0;
        for(k = 0; k < matrixA.width; k++) {
            sum += matrixA[i][k] * matrixB[k][j];
        }
        matrixMult[i][j] = sum;
    }
}
```

First, two matrices of height and width n are defined and filled with random integer numbers from 0 to 100. Also an empty matrix for the result is initialized. For each field of the resulting matrix one needs to calculate the dot product of row i from matrix A and column j of matrix B. This is achieved by multiplying each field of row i from matrix A with each field of column j from matrix B and accumulating the sum over k. The sum is then stored as field i, j in the resulting matrix. The algorithm has a complexity of $O(n^3)$.

2.3.4 I/O Test

The fourth test in this benchmark suite is a disk test. Each serverless cloud provides a temporary file system which the functions can use. The function can for example write a file with some intermediate results which it will later read again for further use. Other function calls which run in the same instance share this file system and can also access the file. Given that serverless tends to be or should be stateless, the feature of a temporary storage is not of great importance. Nevertheless, specific applications might benefit from such a feature and therefore it is included in this thesis. I/O operations are often expensive and in a synchronous function that can lead to CPU wait.

The implementation of the test is fairly simple. Listing 2.4 shows the pseudo code of the test.

Listing 2.4: I/O test pseudo code

```
text = ""

for(i = 0; i < s; i++) {
    text += "A";
}

startWrite = Time.Now()
for(i = 0; i < n; i++) {
    writeFile(i+'.txt', text, 'utf-8');
}
endWrite = Time.Now()

startRead = Time.Now()
for(i = 0; i < n; i++) {
    file = readFile(i+'.txt', 'utf-8');
}
endRead = Time.Now()

writeTime = endWrite - startWrite
readTime = endRead - startRead
```

The algorithm writes files and then reads them. It takes two input parameters n defining the number of files and s the size of each file in bytes. A string `text` is generated with length equal to s. Since the encoding used is `utf-8` each normal character takes 8 bits or 1 byte of storage. Then n many files are written to the file system. After that all the written files are read. Both these operations are timed and the result will be the time it took to write respectively read the files. The complexity of the algorithm is $O(n)$.

2.3.5 Custom Test

Last but not least there is a customizable test. The objective of this test is that the user can easily implement his own function and then benchmark it. The skeleton is provided for each one of the four clouds in each of the four languages. A timer is included to measure the execution time which can later be further analysed.

Chapter 3

Implementation

In this chapter, the implementation of the main application which orchestrates everything will be presented and explained. First, the components and their functions will be briefly explained and later every main task the user can do will be explained in detail.

3.1 Overview

The main application is implemented in JavaScript and uses the Node.js framework. It manages all user input and executes the actions or delegates them to other components. This application is nicely packaged with Docker and runs completely on it. Only the Linux packages `docker-ce` and `docker-compose` are needed to execute this program. Figure 3.1 depicts all components and gives an overview of this benchmarking suite.

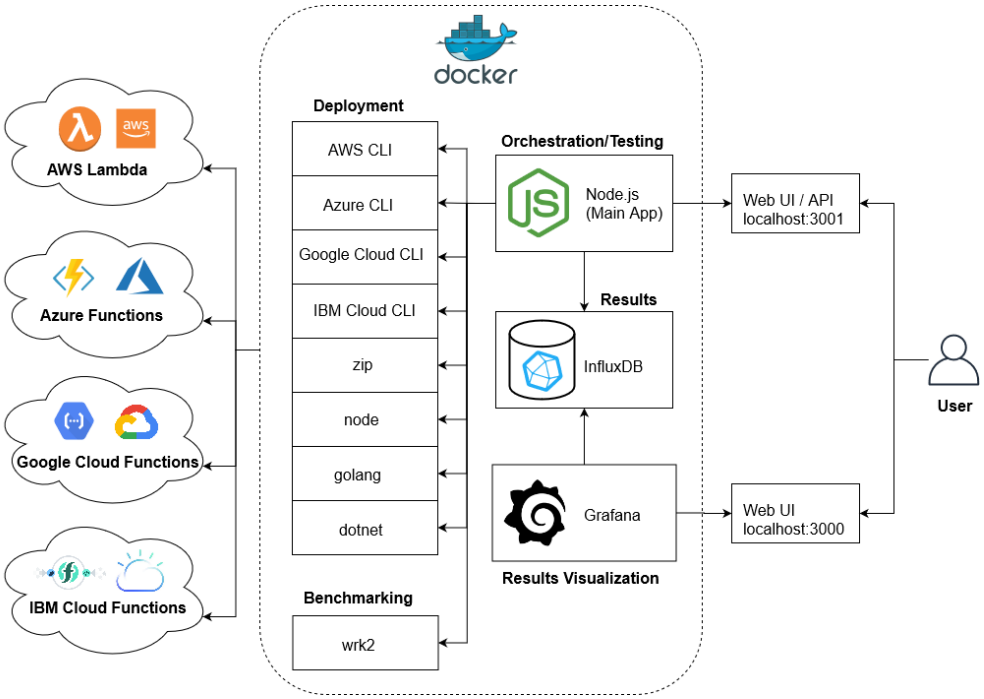

Figure 3.1: Benchmark Suite Architecture. Source: illustration by author

The four clouds and their services can be seen on the left side, the involved Docker images in the middle and they ways to interact with the system can be seen on the right side. Each component will be explained step by step in the following.

- **Main application:** The main application is the core of this benchmark suite and manages the four main tasks: deployment to the clouds, running simple tests, running benchmarks and calculating and estimating prices. It is available to the user through a web GUI and also through an API. In a first step, the user can deploy the five different tests described in section 2.3. Once the tests are deployed, they can be tested and benchmarked. Additionally there is a pricing calculator which calculates the cost for all four clouds either by setting the parameters manually or in regards to a previously executed test.
  Since the main application runs on Docker and invokes other containers which also run in Docker, the main container needs access to `/var/run/docker.sock` i.e. mounting it as volume. Otherwise a container cannot run or start another container for security and integrity reasons. This concept is called Docker-in-Docker and should generally not be used and is more of a hack than a feature. In this case it serves a simple purpose and does not implicate any issues.

- **InlfuxDB:** In this time series database, all the results from tests will be stored for later use in Grafana or pricing calculation.

- **Grafana:** Grafana is an open source tool to display, plot and monitor data stored in a database. The results gathered by the tests and stored in InfluxDB will be displayed in Grafana.

- **CLIs:** Each cloud provides a Docker image of their CLI (with the exception of AWS) to use with Docker rather than to install it on the host. Resources can be deployed, deleted and managed completely by the CLI and no interaction on the web portal is needed.

- **Runtimes (node, golang, dotnet):** In addition to the source code, a built and packaged zip file is needed for the deployment on all cloud except for Google. Therefore these three images are necessary in order to install packages and build the source code for Node.js, Go and .NET.

- **zip:** This simple image zips the files that need to be deployed to the clouds. Therefore there is no need to have zip installed on the machine the benchmark suite runs.

- **wrk2:** wrk2 is a powerful HTTP benchmark used in this application. For simplification it is also containerized.

3.2 Requirements

To use this application, there are some prerequisite steps necessary. First of all, the user needs to have or create accounts on the clouds he intends to test on. This process will be not described in detail since it is straightforward to do. There is however a small guidance for each cloud provider in the documentation on GitHub.
Secondly, the user needs to have a Docker environment installed. This is not too difficult on Linux and described exemplary for Ubuntu 18.04 on GitHub.
After these steps have been completed some more manual initialization and configuration steps are required from the user. He needs to create a few Docker volumes and perform the login process for each CLI belonging to the cloud intended to be tested.
As last point some free storage is required because some Docker images are quite large, in total around 5.15 GB. In addition, some free space should be reserved for the data that will be stored in the database, 1 GB should be sufficient. Table 3.1 shows all images and their sizes.

Repository	**Tag**	**Size**
bschitter/benchmark-suite-serverless-computing	0.1	358MB
mikesir87/aws-cli	1.16.310	186MB
google/cloud-sdk	274.0.1-alpine	287MB
mcr.microsoft.com/azure-cli	2.0.78	1.04GB
mcr.microsoft.com/dotnet/core/sdk	2.2-alpine3.9	1.48GB
bschitter/alpine-with-zip	0.1	6.32MB
bschitter/alpine-with-wrk2	0.1	232MB
ibmcom/ibm-cloud-developer-tools-amd64	0.20.0	309MB
golang	1.11-stretch	757MB
node	10.16.2-alpine	76.4MB
grafana/grafana	6.3.2	254MB
influxdb	1.7.7-alpine	137MB
Total size		**5.15GB**

Table 3.1: Docker images

3.3 Deployment and Cleanup

This section will describe the deployment and the cleanup process.
After the application has been started, the user can use the web interface (exposed on port 3001) to take actions. The first thing he needs to do is deploy a test. The figure 3.2 shows a screenshot of the web interface.

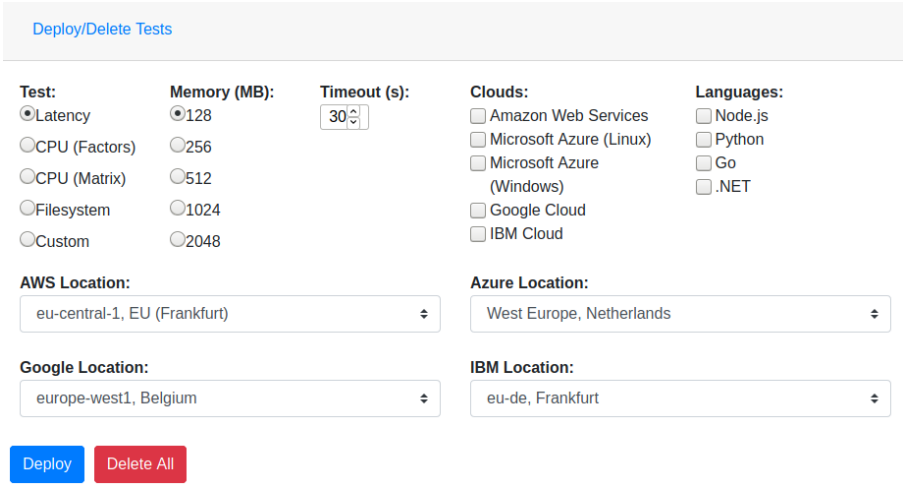

Figure 3.2: Web GUI - Deploy/Delete Tests, source: screenshot of the application

The following paramters can be chosen:

- **Test:** The test that will be deployed (as described in section 2.3).
- **Memory:** The amount of memory the function will have available.
  Remark: Not applicable for Azure as memory is assigned dynamically at a maximum of 1536 MB.
- **Timeout:** The time limit after which a running function will time out.
  Remark: Not applicable for Azure since it handles timeout configuration in a configuration file.
- **Clouds:** The cloud providers the tests will be deployed on, multiple choices possible.
- **Languages:** Runtimes respectively languages to deploy the function in, multiple choices possible.
- **Locations:** The region where the function will be deployed to.

Next the *Deploy* button can be pressed and the application will initiate the deployment. On the right hand side of the web interface there will be information about the progress. The deployment process is highly parallelized (if possible) to make it as fast as possible. In the appendix A, flow charts are illustrating the deployment and cleanup process for each cloud. For a cleanup the user can invoke the *Delete All* button. Considering the cleanup is implemented very minimalist it will delete all deployed Lambda functions and API gateways on AWS and IBM, all resource groups with a name containing latency, factors, matrix, filesystem and custom and on Google all functions in the configured project. It should therefore be used very carefully and ideally with separate accounts only for this purpose.

3.4 Testing

In this section it is briefly explained how to execute a test and see its results. The test will send every five seconds a request to the functions previously deployed to see how they perform under a low, constant load. As options, a name for the test and the functions parameter can be set. The emerging result will serve as a starting point for comparison to the benchmark and optimally the function should deliver the same performance under heavy load. Figure 3.3 illustrates a screenshot of Grafana containing test results of a latency test in Node.js. At the top, there are three drop down menus to choose the test type, the test name (given at the start of the test) and the data points interval.

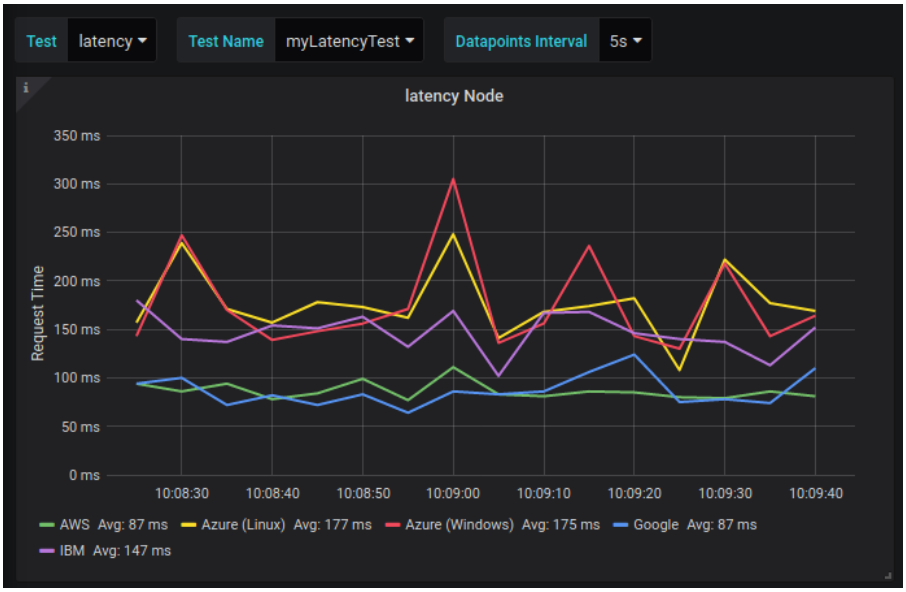

Figure 3.3: Test results in Grafana. Source: screenshot Grafana

The plots can be grouped by cloud provider or by runtime. There is a predefined Grafana dashboard for both options, the user can switch depending on his interests. An example result of a general test can be seen in section 4.1.3.

3.5 Benchmarking

The benchmarking part mainly relies on the wrk2 image. The parameters can be set in the web interface but they are basically just forwarded to wrk2. For the benchmark itself the user can choose the following parameters: requests per second, duration of the benchmark and the desired test to run e.g. the CPU factors test. Afterwards the load test will start and benchmark the chosen function on each cloud and runtime it was deployed to. This process has to run sequentially otherwise the host of the benchmark suite could potentially not handle the load. After the test has completed results will be parsed and inserted into the database and can be viewed as a table in Grafana. Figure 3.4 shows an excerpt of a result.

Provider	Language/Runtime	Latency (Avg.)	Latency (Max.)	Latency (StDev.)	50th Percentile	95th Percentile	99th Percentile	Requests/second	Requests/second (Goal)	Sample Count	Errors
aws	python	41.08 ms	288.26 ms	12.28 ms	40.86 ms	61.89 ms	83.78 ms	99.14 reqps	100 reqps	-	0
ibm	python	462.19 ms	1.62 s	215.23 ms	433.66 ms	845.31 ms	1.06 s	99.05 reqps	100 reqps	-	1
google	python	169.07 ms	516.35 ms	78.92 ms	174.08 ms	293.38 ms	377.60 ms	99.14 reqps	100 reqps	-	7
ibm	node	747.02 ms	4.95 s	629.09 ms	486.40 ms	1.93 s	2.59 s	98.87 reqps	100 reqps	-	13
google	node	180.14 ms	3.85 s	110.44 ms	167.93 ms	303.87 ms	395.77 ms	99.14 reqps	100 reqps	-	3
google	go	167.06 ms	4.46 s	199.35 ms	131.33 ms	316.67 ms	707.58 ms	99.11 reqps	100 reqps	-	6

Figure 3.4: Benchmark results in Grafana. Source: screenshot Grafana

A more detailed test example with increasing load is discussed in section 4.1.4.

3.6 Pricing

With this component of the application one can calculate hypothetical prices by entering number of invocations, execution time per call, size of the return body and allocated memory. The prices for all clouds will be calculated and displayed in a table. The functionality is basically the same as the pricing calculators provided by the cloud providers, except here all is in one place and directly comparable.
Furthermore, the calculator takes the performed tests into account. One can select a previously run test, select a runtime and then one only needs to provide the estimated number of invocations per month. The calculation will happen by taking the execution time from the test results which of course can vary quite a lot between cloud providers and runtimes. This method allows a much better approach of estimated cost. In section 4.2 the same example will be explained and calculated for both hypothetical and with actual test results as input.

Chapter 4

Results

In this chapter results of performed tests will be presented and explained, an example for a pricing calculation will be given, the services among the cloud providers will be compared and some general advantages and disadvantages about serverless computing will be specified derived from these results.

4.1 Tests

In this thesis, four different tests have been performed. Each one will be discussed in detail and results will be presented in the following subsections.

4.1.1 Latency Test

The objective of the latency test was to measure the latency for all clouds and regions. This test differs from a normal ping, that effectively a cloud function is executed instead of just returning a message on a more abstract level. The test was performed in Node.js with 128MB of memory (Azure 1.5GB) with the latency test described in section 2.3.1. Every 5 seconds, a requests was sent to each cloud and region until a sample size of 100 was obtained. This test was carried out from Bern, Switzerland.

Remarks.

- For Azure, this test was only done with Linux as underlying OS. The OS should not matter in regards to latency.
- The *westus* region of Azure had problems deploying it in Node.js and therefore .NET was used. Following error was produced: `The scale operation is not allowed for this subscription in this region. Try selecting different region or scale option.`
  Also trying to create the function in the Azure portal failed and an error was indicated but the error message could not be viewed.
- On Azure region *australiaeast* the function could not be deployed, although no error was risen. Therefore it was deployed on Windows instead of Linux.

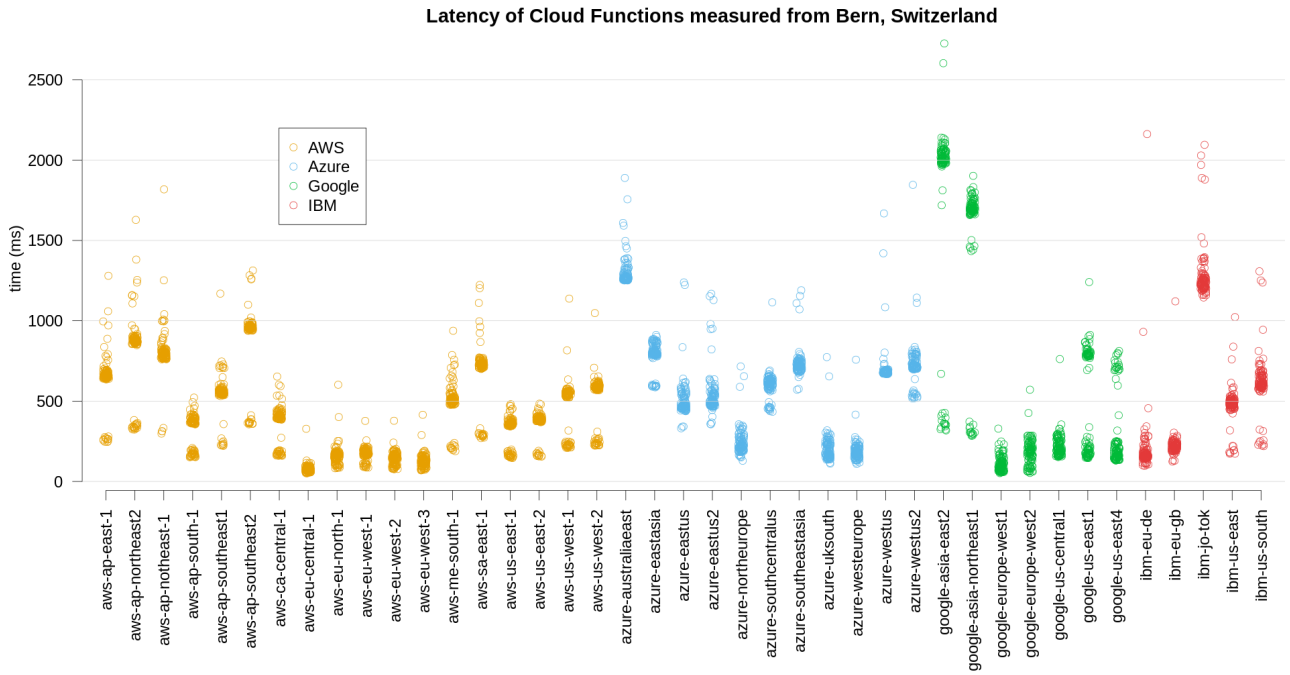

Figure 4.1: Latency test scatter plot,
Source: illustration by author

The graphic 4.1 shows a scatter plot with the results of the test.
For more detailed results see table B.2. The raw result file, R script and plot image are also on GitHub.

4.1.2 Cold Start Test

This test measured the cold start latency. A cold start happens when the function takes longer to start up and run than usual. This occurs mostly after the deployment, after the function has not been used in a while or when a new instance needs to be provisioned for scaling purposes. In order for the cloud provider to get up the instance and deploy the specific code on it, some time can pass. Figure 4.2 shows what happens on Azure when a cold start is required and when the app is already warm.
As indicated in the figure, several steps are required initially before the function can execute. A server has to be allocated, the worker is set up with the relevant code, packages and extensions are loaded (if needed), the function gets loaded into memory and then finally it can run. If the function is warm and no cold start happens, it is ready and can just be invoked. This process is similar in other clouds.
The cold start latency has been tested ten times for each runtime on each cloud. Memory size was defined with 512 MB and no packages were loaded into the function. The test was realized in the following regions: AWS on `eu-central-1`, Azure on `westeurope`, Google on `europe-west1` and IBM on `eu-de`. However, the region should not have any implications on the cold start latency. The cold start latency was calculated the following way:

$$\text{Total Request Time} - \text{Normal Average Latency} = \text{Cold Start Latency}$$

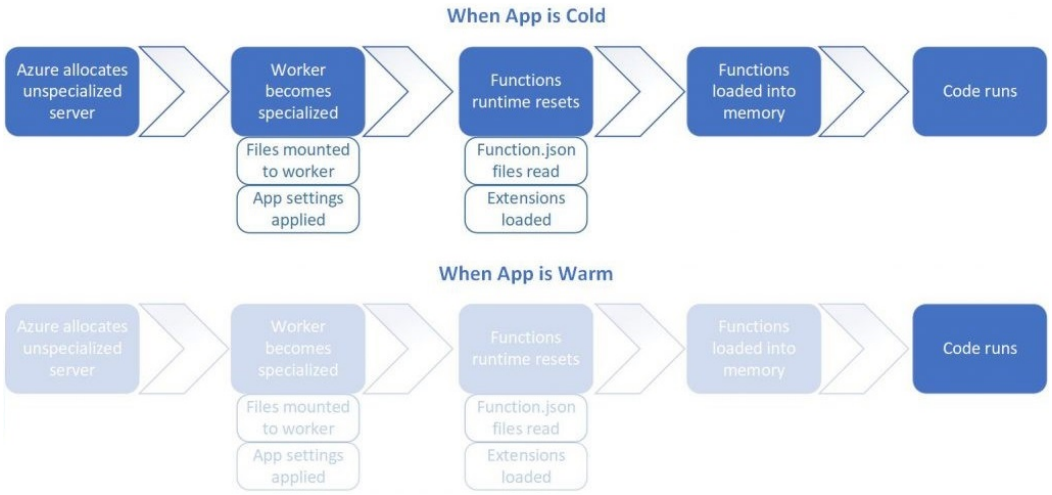

Figure 4.2: Cold and warm start on Azure. Source: Azure [Tre18]

Figure 4.3 shows the result of this test. A box plot shows the cold start latency for every cloud service provider and runtime. AWS is overall the fastest, with an average cold start latency of only 335 ms for Node.js, Python and Go. For .NET it is much higher with an average of 1739 ms, possibly due to the nature and compilation of .NET respectively C#. On Azure, cold start latency is always more than 2 seconds and ascends up to 5 seconds, except for .NET on Windows which has an average cold start latency of 1917 ms. Google has compared to AWS and IBM relatively high cold start latency around 2 to 3 seconds. IBM shows a similar pattern as AWS although the cold start latency is around 600 ms higher for Node.js, Python and Go but similar for .NET.

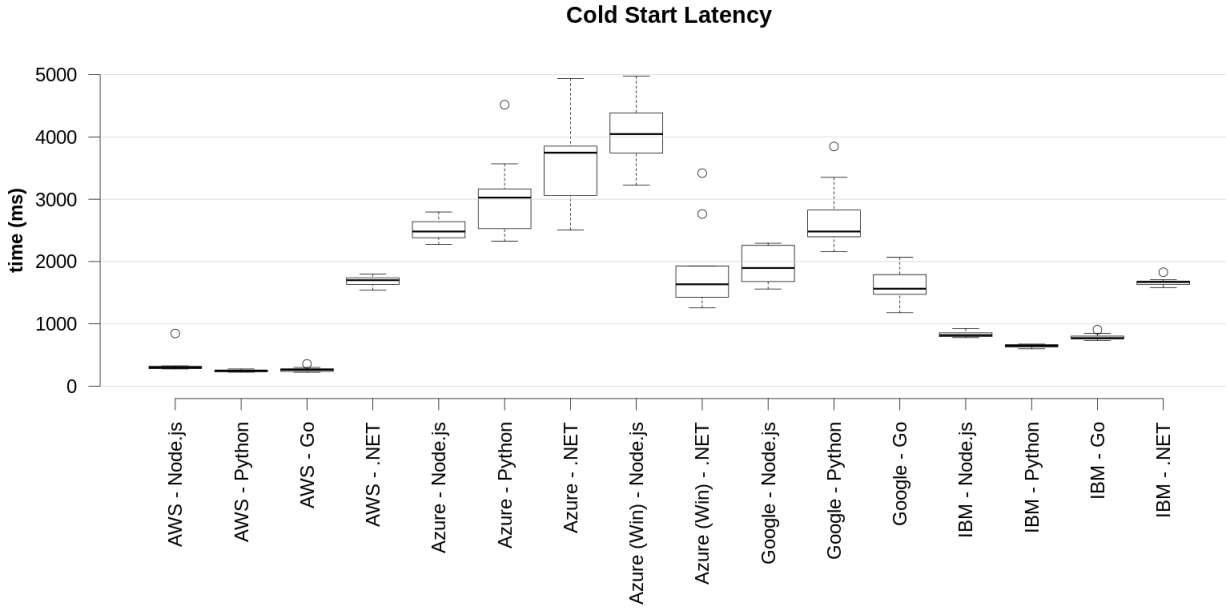

Figure 4.3: Cold start latency box plot. Source: illustration by author

On AWS and IBM it took usually around 10 minutes of no activity for the instance to be recycled by the provider, on Azure around 10-20 minutes and on Google it varied from 10 minutes up to 10 hours.

4.1.3 General Test

This test was performed to analyze the functions under normal circumstances, meaning that there is not much load respectively that the load can be handled well by the cloud provider. The objective is to compare the achieved performance of the clouds regarding execution speed and the thereby implicated costs. For this evaluation the CPU factors test (see section 2.3.2) was executed. The function was invoked every five seconds until a sample size of $n = 100$ was reached. This for each MB configuration, each runtime and each cloud. As input parameter 26′888′346′474′443 was selected since with this number the function finished ahead of the timeout for low configurations and simultaneously was not too quickly finished for high configurations. Figure 4.4 illustrates the result of this test in Python in a scatter plot.

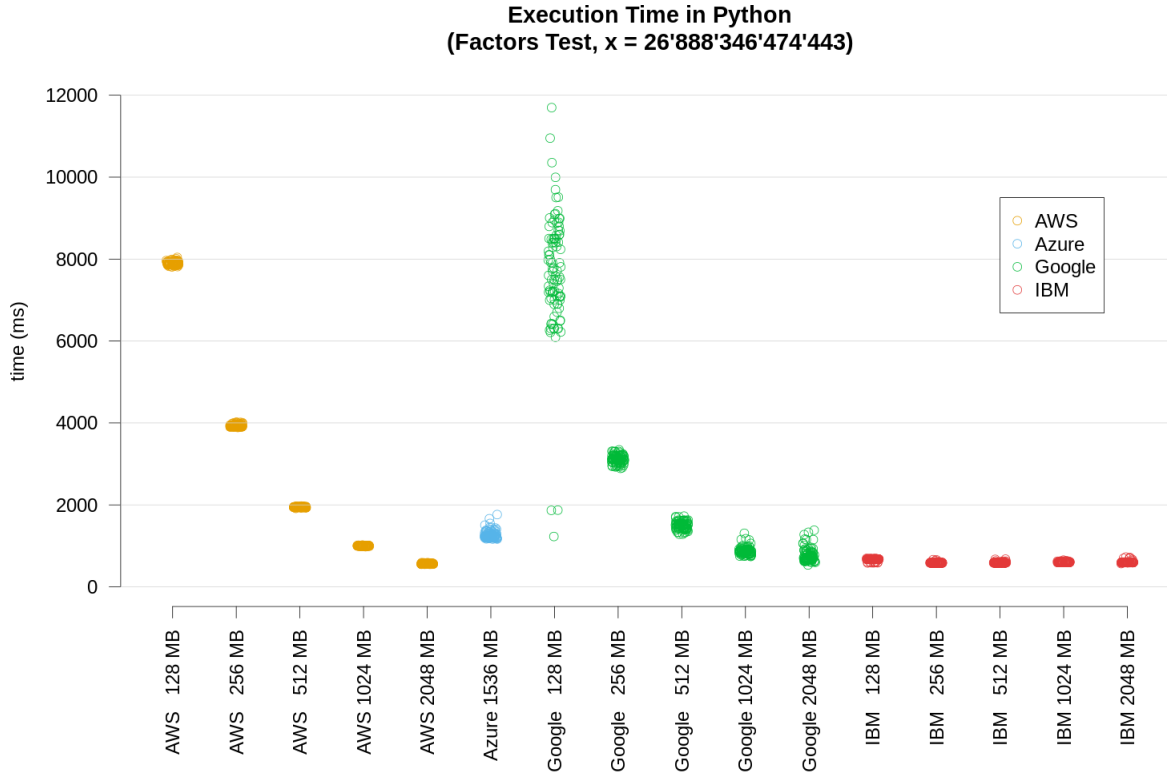

Figure 4.4: Scatter plot of execution times of the CPU factors test in Python
Source: illustration by author

A few interesting characteristics can be observed. On AWS Lambda everything seems as it should be and is hence a good showcase. Firstly, the standard deviation is very low and therefore the execution time extremely consistent. With every doubling of the allocated memory the execution time is halved, as it should be since CPU performance scales linearly to allocated memory, according to Amazon [AWSb]. Conforming to Azure's

memory configuration of 1536 MB its performance should be between 1024 and 2048 MB of the competitors performance. However, it is even slower than 1024 MB instances of the other clouds. The results on the Google Cloud Platform follow a similar pattern as on AWS, but they are more scattered especially for 128 MB of memory. Moreover Google is faster than AWS for 128, 256, 512 and 1024 MB but not for 2048 MB. For the first three times doubling the memory leads to halved execution time. But not entirely for the next two steps since Google does not scale CPU completely linear to allocated memory [Gooc]. At last, IBM shows a very strange pattern. It seems that memory allocation does not correlate with CPU allocation in any way. All five different sizes perform practically the same. This is remarkable since the pricing model of IBM only accounts for GB-Seconds used. This means that a user could deploy his application always with the smallest memory option possible and would thereby get the same performance for the smaller price.
The results and plots for this and the three other runtimes can be found on GitHub.

4.1.4 Load Test

The load test is designed to benchmark the serverless functions up to 1000 requests per second. This is carried out with the HTTP benchmark tool *wrk2* (see [Ten19]). A function is first called with 10 requests per second for the duration of 1 minute. Subsequently with 25, 50, 100, 200, 400, 800 and finally 1000 requests per second each time for one minute. In between is a short break of 10 seconds to allow the function to process requests that are still queued. As test, the matrix function with a parameter of 100 is used. With this setup the matrix function should have an average execution time of about 100ms for Node.js, Go and .NET and around 250ms for Python. The graphic 4.5 displays the average latency results grouped by runtime.

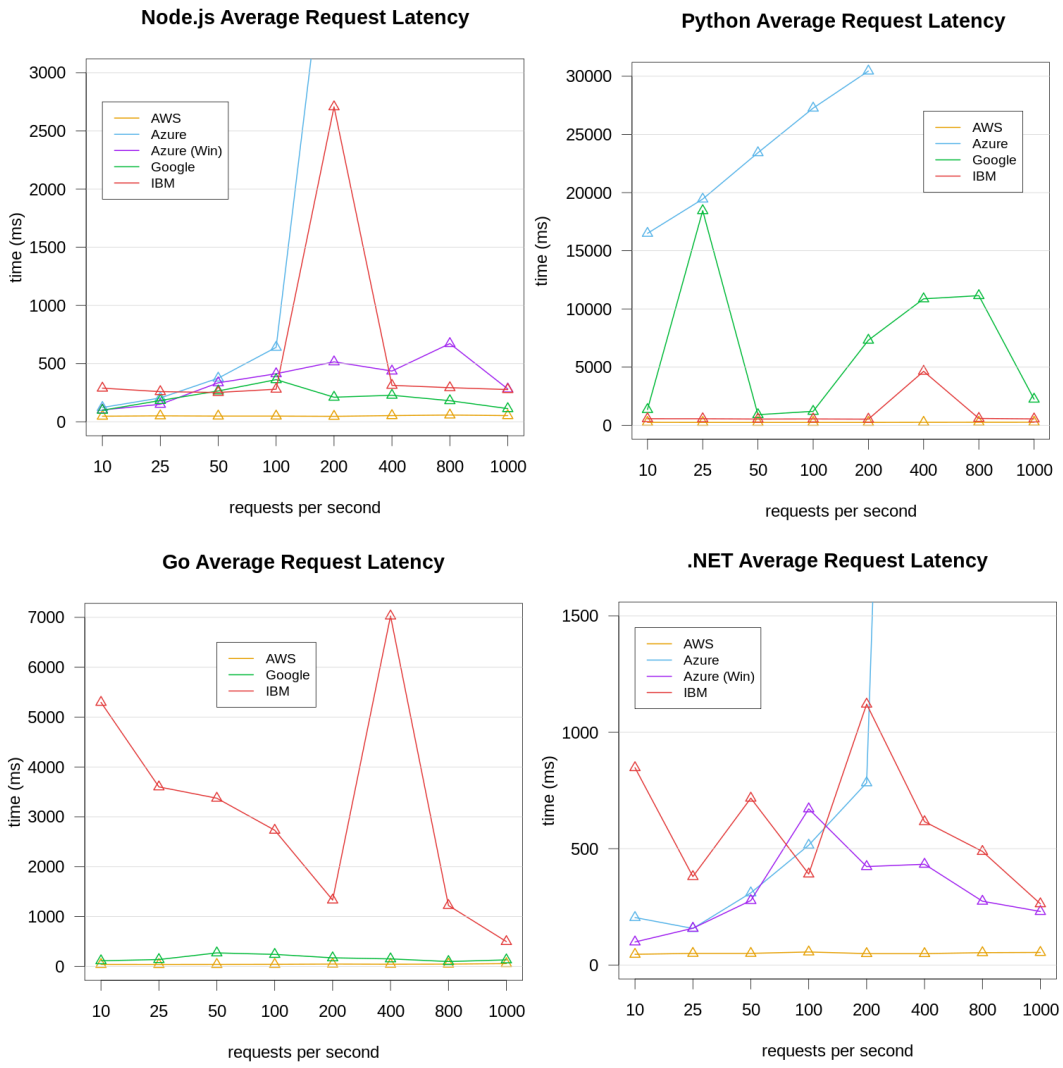

Figure 4.5: Load test average latency. Source: illustration by author

The results are now discussed step by step. AWS has for all four runtimes an extremely steady response latency throughout the increasing load. This is very remarkable and a very good result and can be probably traced back to the low cold start latency and the good and consistent performance presented in section 4.1.3.

On Azure it looks quite different. Azure can handle the load on Windows as operating system. But in the first increasing steps from 10 to 100 or 200 RPS latency rises although it flattens out afterwards. On Linux, Azure performs very bad and can not scale out quickly enough to serve the requests in an acceptable time frame. The response time climbs more or less linearly to the amount of request per second sent. That is a strong indication that none or too few new instances are allocated to handle the load. At the time of 1000 requests per second one could see in the Azure portal Live Metrics Stream that e.g. for .NET only 12 instances have been deployed (see figure 4.6). Additionally the screenshot shows that only 500 requests per second get handled by the function, the rest is probably waiting in a queue.

Remark. In the screenshot the request duration differs from the wrk2 results since that is most likely only considering execution time without queuing time.

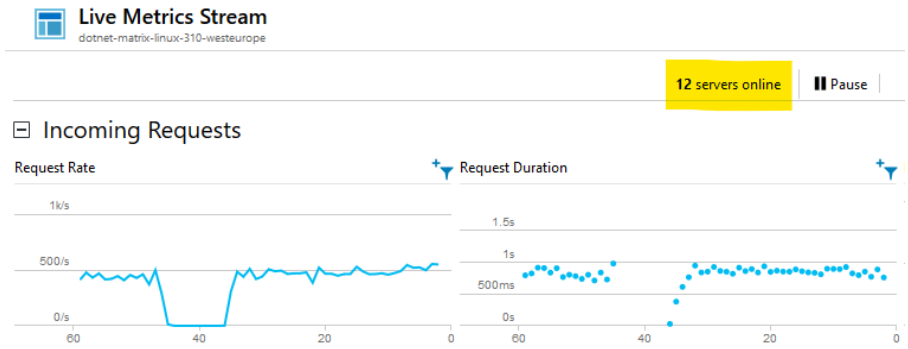

Figure 4.6: Azure Linux .NET Live Metrics Stream during 1000 RPS
Source: screenshot Azure portal

For Node.js and .NET the latencies went up to 24s respectively 18s and are not plotted in the graph for illustration purposes. On Python, Azure returned from 200 RPS and above almost only HTTP errors and the result of wrk2 is not representative and hence not plotted.

Google can manage the load generally well, although latency increases a little bit for Node.js and Go. With Python, Google has the biggest problems and seems overwhelmed when requests increase. It can not scale quickly enough and the average request duration increased from 1356ms to 18448ms with 10 RPS respectively 25 RPS. After that, this phenomena happens again but less drastically. This can possibly be explained by the fact that the function has a longer execution time in Python and Google can scale longer running functions less good. Overall the performance is good. Figure 4.7 shows the number of instances allocated during the Go load test on Google.

Lastly, the result of IBM will be discussed. Those outcomes differ from the others relatively much. On Node.js and .NET it can roughly keep up to the competition but operates a little slower and on Node.js it has a striking outlier. The Python function on IBM nearly performs as well as on AWS, but the same cannot be said about Go. The Go runtime performed much slower in the load test as with the competitors, although as seen in the cold start test in section 4.1.2 it has a similar cold start latency as Node.js and Python and as shown in the

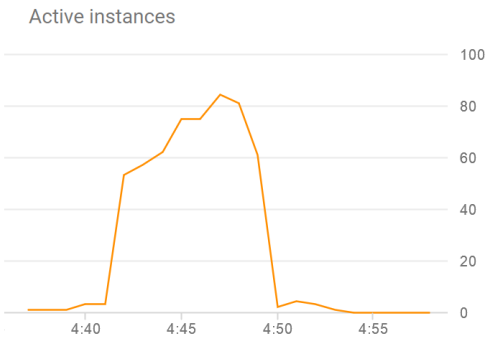

Figure 4.7: Google Go active instances during load test
Source: screenshot Google cloud portal

appendix table B.3 it performs generally well on Go. Thus this bad performance cannot be derived from this or other tests and it is possible that IBMs auto scaling mechanism does not work well for Go. Of the amount of instances deployed, IBM delivers no statistics.

So far, the latency has been examined and now the actual throughput will be explained. Most of the cloud providers and runtimes could handle the load and only had an irrelevant smaller throughput than tested against. Therefore only significant impacts in throughput will be presented, meaning that the throughput was less than 90% of the tested one. The subsequent table 4.1 shows where throughput was below 90%. In addition, there are detailed result plots (figure B.1 - B.4) and tables (table B.4 - B.7) in the appendix.

Cloud	Runtime	RPS (goal)	RPS (actual)	RPS in %
Azure	Node.js	200	179	89.50%
Azure	Node.js	400	227	56.75%
Azure	Node.js	800	278	34.75%
Azure	Node.js	1000	327	32.70%
Azure	Python	10	6	60.00%
Azure	Python	25	12	48.00%
Azure	Python	50	18	36.00%
Azure	Python	100	25	25.00%
Azure	Python	200	30	15.00%
Azure	Python	400	15	3.75%
Azure	Python	800	14	1.75%
Azure	Python	1000	15	1.50%
Azure	.NET	400	324	81.00%
Azure	.NET	800	407	50.88%
Azure	.NET	1000	512	51.20%
Google	Python	25	12	48.00%
Google	Python	200	168	84.00%
Google	Python	400	297	74.25%
Google	Python	800	575	71.88%
IBM	Go	25	21	84.00%

Table 4.1: Load tests that achieved less than 90% of RPS

4.2 Pricing

In this section two pricing examples will be calculated and explained. The first one will be theoretical and the second one will be based on the general test in section 4.1.3. Table 4.2 shows the prices per unit for each cloud. Prices on AWS and Azure can vary depending on the location. The table and the examples use the prices of the region `eu-central1` for AWS and `westeurope` for Azure. All clouds round up GB-Seconds and GHz-Seconds up to the next 100ms per invocation, except for Azure which rounds it up to the next 1ms.

	per invocation	**per GB-Second**	**per GHz-Second**	**per GB network egress**
AWS	0.0000002	0.000016667	-	0.09
Azure	0.0000002	0.000016	-	0.087
Google	0.0000004	0.0000025	0.00001	0.12
IBM	-	0.000017	-	-

Table 4.2: Serverless functions pricing, all prices in USD as of 22.01.2020
Data source: [AWSd, Micb, Gooc, IBMc]

As one can see Google seems to be to have the most sophisticated pricing model of all. Charging exactly for not only the GB-Seconds used but also the GHz-Seconds. The pricing model of AWS and Azure are very similar and the one of IBM only takes into account GB-Seconds and they are not significantly more expensive.
All of the cloud have a certain free tier quantity which is included per month. Table 4.3 lists the free quantities. Network egress free tier is shared with all network egress of the cloud. Free tier is not considered in the following two examples.

	Invocations	**GB-Seconds**	**GHz-Seconds**	**GB network egress**
AWS	1'000'000	400'000	-	1
Azure	1'000'000	400'000	-	5
Google	2'000'000	400'000	200'000	5
IBM	-	400'000	-	-

Table 4.3: Serverless functions free tier as of 22.01.2020
Data source: [AWSd, Micb, Gooc, IBMc]

Example 1

Let's assume a company has a custom application for image processing written in Python. It does some modifications to the image and then saves the result in a storage solution of the correspondent cloud. Each month around 10 million images are processed. This task is not time critical but should be cost efficient. On a developers desktop computer the task takes around 5 seconds to complete and uses up to 450 MB of memory. On a cloud platform a similar result is expected. In this example there is no network egress.

Calculation

Cloud :	**Invocations**	+	**GB-Seconds**	+	**GHz-Seconds**	=	**Cost**
AWS :	$10M \cdot 0.0000002\$$	+	$10M \cdot 0.5 \cdot 5 \cdot 0.000016667\$$			=	418.675$
Azure :	$10M \cdot 0.0000002\$$	+	$10M \cdot 0.5 \cdot 5 \cdot 0.000016\$$			=	402.000$
Google :	$10M \cdot 0.0000004\$$	+	$10M \cdot 0.5 \cdot 5 \cdot 0.0000025\$$	+	$10M \cdot 0.8 \cdot 5 \cdot 0.00001\$$	=	466.500$
IBM :		+	$10M \cdot 0.5 \cdot 5 \cdot 0.000017\$$			=	425.000$

As one can see the cheapest in this case would be Azure, followed by AWS, IBM and finally Google as the most expensive. With that amount of invocations and execution time all clouds are comparable.
Now the problem with this calculation is that it only assumes the execution times on the clouds which is the most essential part of the pricing calculation. Some clouds may perform much better than others. Example 2 will take actual execution time into consideration.

Example 2

This example takes the result of the CPU factors test of section 4.1.3 as a basis. This test is also in the runtime Python. Let's also assume 10 million function calls are invoked per month. With the fixed parameter of $26'888'346'474'443$ the function should not consume more than 100 MB of memory. Its return size is around 4 KB per call. As execution times we take the average (rounded up to 100ms if applicable) of the result.

Remark. Azure claims that it only charges the GB-Seconds the function actually used, rounded up to the next 128 MB step [Micb]. It seems therefore that the pricing model of Azure is similar to the one of IBM, delivering the same performance independent of memory size.

Doing the equivalent calculations as in example 1, the following results are obtained as shown in table 4.4.
AWS is an exemplar in showing how the pricing works. Since with each doubling of memory execution time halves the price is exactly the same because also the GB-Seconds remain the same. But with 170$ to 200$ it is not the cheapest in the list.
Azure strikes out to be very cheap. Since this example function application only uses 100 MB Azure also only charges for 128 MB. However the actual metrics of how much memory was used does not exist for Linux and is only depicted in a graph for Windows. But in the case this test is correct the cost-performance ratio is very good for low memory applications.
Google generally conforms with the pricing results of AWS. It positions itself in the same price region although it climbs in costs at 128 MB and 2048 MB. This can be traced back to the fact that it performed inconsistent and relatively seen slower for 128 MB, and for 2048 MB not much performance was gained. Therefore these two options are more expensive.
IBM leans on Azure's method of calculating prices. It only charges for the defined and allocated GB-Seconds,

	Rounded time	Invocation	GB-Seconds	GHz-Seconds	Network	Total
AWS 128MB	8000ms	2.00$	166.67$	-	3.43$	172.10$
AWS 256MB	4000ms	2.00$	166.67$	-	3.43$	172.10$
AWS 512MB	2000ms	2.00$	166.67$	-	3.43$	172.10$
AWS 1024MB	1000ms	2.00$	166.67$	-	3.43$	172.10$
AWS 2048MB	600ms	2.00$	200.00$	-	3.43$	205.43$
Azure 128MB	1267ms	2.00$	25.34$	-	3.32$	30.66$
Google 128MB	7700ms	4.00$	24.06$	154.00$	4.58$	186.64$
Google 256MB	3200ms	4.00$	20.00$	128.00$	4.58$	156.58$
Google 512MB	1600ms	4.00$	20.00$	128.00$	4.58$	156.58$
Google 1024MB	900ms	4.00$	22.50$	126.00$	4.58$	157.08$
Google 2048MB	800ms	4.00$	40.00$	192.00$	4.58$	240.58$
IBM 128MB	700ms	-	14.88$	-	-	14.88$
IBM 256MB	600ms	-	25.50$	-	-	25.50$
IBM 512MB	600ms	-	51.00$	-	-	51.00$
IBM 1024MB	600ms	-	102.00$	-	-	102.00$
IBM 2048MB	700ms	-	238.00$	-	-	238.00$

Table 4.4: Pricing example regarding test results

but performs basically always the same. Hence, it gets approximately linearly more expensive with more memory allocation.

4.3 Evaluation of all services

In this section. the four serverless services will be compared. Two aspects were crucial for the evaluation: First, the previously discussed test and benchmark results and secondly, my personal opinion which was developed during this research and its implementation.

Amazon Web Services Lambda

Overall, AWS was astonishing with its performance. It has by far the lowest cold start latency and is very consistent regarding execution time. Furthermore it could handle the increasing load test without any issues. The request time did not increase significantly and the desired amount of requests per second was nearly achieved. Also the management of function deployments with the CLI or in the portal worked flawlessly. At first, it is a little tricky to set up a function with a trigger (especially with the CLI, see figure A.2) but the documentation is good and there are many examples on the internet.
Following is a list of some small problems or aspects that could be improved:

- No official Docker image for the AWS CLI. There are other prebuilt ones that can be used or it is easy to make your own.
- To delete functions, one needs to invoke the command `aws lambda list-functions` in the CLI. Sadly, this command only takes into account one region and does not allow to retrieve multiple regions at once.

- For security reasons one can only delete every 30 seconds an API gateway. There is no possibility to remove or configure this restriction.

Microsoft Azure Functions

Azure cannot compete to Amazon in terms of performance and usability. The cold start latency is around two to four seconds and possibly therefore it does not scale that well (at least on Linux). As a result, load tests were good on Windows but very bad on Linux. Normal or low use performance is nonetheless okay.
Using the portal can be frustrating since navigating through it or deploying something can be slow. In general, the CLI works good but there are some actions that can only be done on the portal and sometimes a specific deployment command failed (see this issue on Github https://github.com/Azure/azure-cli/issues/10574). Additionally to the CLI, Azure Functions Core Tools is a required Node.js program to run and test the functions locally and ideally they are also deployed with this program. This tool however is not included in the CLI and there is also no Docker image available nor was I able to create one that worked. There is an alternate way to deploy the function (which was used in this thesis) with a zipped package that was built before.
From time to time various documentations of Microsoft were contradicting each other or not up to date.
Azure has on the portal a nice *Live Metrics Stream* where one can monitor the functions. It shows real time data (only about 1-2 seconds behind) such as requests per second, execution time per request, CPU utilization and the number of servers which are allocated (see figure 4.6).
Considering that Azure has three different function generations and three different execution plans it makes the impression that the service architecture has grown over time and is thus a little chaotic.
Some more prospects that can be improved are:

- The name given to the function app has to be unique since it forms part of the invocation link. This can lead to unnecessary deployment errors if the process is automated.

- Azure functions generation 3 did not work at all. I tried upgrading from generation 2 to 3 when it became generally available but the parameter for generation 3 in the configuration file was simply not considered by Azure. This setting also can not be changed afterwards with the CLI, only in portal. Not at all for Linux environments since they are read only when deployed with the CLI.

Approximately 80% of debugging and fixing the application can be traced back to Azure behaving badly or being vastly different than the other three cloud service providers.

Google Cloud Functions

The Google Cloud was very satisfactory to use. The portal is simple and clearly structured and the CLI handled well. Where the other clouds need multiple commands to get a function deployed and running it is just one simple command with Google (see figure A.3). The documentation is complete and clear.
In regards to performance Google functions is good. The cold start latency is relatively high compared to AWS and IBM. Execution times are similar as on AWS and scaling during the load test was okay. It seems to work

well for fast executing functions but not so good for slower functions (higher than 0.5 seconds) which could be a deal breaker. On paper Google is the most expensive of them all.
In the Google portal there are some nice graphs to monitor number of allocated instances, execution time, number of invocations and memory used per call. It is not properly real time as the one of Azure since it lags behind a few minutes.

IBM Cloud Functions

Working with the IBM Cloud was a little bit tougher than the others. The CLI documentation is not that straightforward and structured as the ones of the competition and the CLI itself can only be used if there is cloud foundry support for the corresponding region, which luckily is the case for all public regions except Tokyo, Japan.
IBM comes in second place in respect to cold start latency. It has very good performance even with low memory configurations. However it could not keep up to AWS in the load tests and was pretty bad for the Go runtime. A few other remarks:

- IBM cloud functions has a 3000 requests per minute limit, however it is not mentioned in the documentation [IBM19c]. In order to increase this limit a request with a business case has to be submitted to IBM and if reasonable they will increase the limit. I have discovered this limit only during the benchmark test.
- The CLI won't be able to load resources (i.e. `ibmcloud fn api list`) after it hasn't been used for some time. I was not able to figure out the problem. The second try however will work.
- IBM only charges per memory used per time unit (GB-Seconds) but delivers as seen in figure 4.4 always the same performance. This was discussed in section 4.1.3 and in my own opinion this pricing model can be exploited and is not well balanced (counterexample Google [IBMc, Gooc]). However further testing would be necessary to confirm this behaviour.

Discussion and Comparison

In the following, some of the most important aspects to be considered when using serverless platforms will be summarized.

- It is very important to test the function carefully before going into production. Measuring the execution time, optimizing the code as much as possible and deciding which instance size fits best for the function.
- Test the scaling mechanism well and according to the request pattern of the application. Otherwise a big surprise could be right around the corner.
- Calculating prices in theory is okay but it is much more accurate to actually measure execution times of the functions and calculate the prices regarding these times.

Table 4.5 shows a summary and comparison of the key features and general advantages and disadvantages each cloud provider has to offer.

	AWS	**Azure**	**Google**	**IBM**
Performance	very good and consistent	good	good, not consistent for 128MB	good
Cold start	223 - 1798 ms	1256 - 4974 ms	1178 - 3847 ms	599 - 1829 ms
Scaling	very good, response time does almost not increase	good on Windows, very bad on Linux	generally good, not for Python or longer running functions	not that good, high spikes are possible
CLI	good and stable, some improvements possible	slow, unexpected errors happen often	very good and stable, also fast	good, sometimes unexpected behaviour
General pros	Everything handles and performs good	Live Metrics Stream, relatively cheap	Nice graphs for monitoring, sometimes a functions gets executed on a faster instance	Low memory configuration have same performance as big ones, therefore cheap
General cons	Not great monitoring on the portal, the user has to work with randomly generated IDs in the CLI	Portal is slow, CLI usage is only okay, unique naming is necessary for the function app name	More expensive, not in that many regions available	Poor CLI documentation, not in that many regions available

Table 4.5: Summary and comparison of services

4.4 Advantages and disadvantages of serverless computing

This thesis has shown that serverless computing in the form of FaaS can deliver good performance at a reasonable price. For the end user it is fairly simple to start with, to use and maintain compared to VMs or even physical servers. With serverless, companies can focus on building their business logic instead of maintaining operating systems and servers which requires expertise and is especially for smaller companies not always affordable. The probably most important point is auto scaling. If the serverless platform can handle that well it takes away one big crucial factor the user does not longer have to care about. Today there are other solutions for automatically scaling applications (i.e. Kubernetes cluster) but using a serverless platform is definitely simpler and more comfortable.

Nevertheless, there are also downsides of using serverless platforms. A first aspect are the limitations given by the cloud provider. Technical limitations such as available runtimes and programming languages, supported triggers, the possibility of third party services integration, quotas, maximum memory etc. can normally not be changed or influenced by the user. In most cases and particularly for simpler applications this should be no obstacle. Considering the full development stack is managed by the provider users can potentially be forced to

upgrade their application runtime (e.g. Node.js 6 to Node.js 8) when it reaches end of life. If this is announced early enough it doesn't put the users under pressure. But still the action eventually needs to be done and this costs resources. That can be annoying and the user has no control over it.
Another risk of using these services can be a vendor lock-in. If the service is easily capable of being integrated along with other services of the cloud, the user can be entrapped to use them and get unintentionally bonded to the provider. When a change in the application design is on its way or even a change of the cloud provider is considered, these integrations could block a modification in the architecture or a migration.
Furthermore there is no de facto standard on how to implement and deploy the functions. Each cloud uses its custom methods and properties (e.g. request and response object) inside the function which varies across the clouds. There is obviously no common interest of the providers to develop a standardized framework due to their competition.
A last aspect I'd like to mention is efficiency. Since the customer is billed by the execution time of the function it is extremely important to only run highly optimized and efficient code, when executed in high quantities. Otherwise billing for unnecessary or idle CPU time (e.g. run remote operations synchronously) can happen. This concern should be obvious to developers and entrepreneurs but the fact that one is billed exactly by the usage emphasizes it even more (compared of having an over provisioned server).

Chapter 5

Conclusion

5.1 Summary

This thesis and benchmark suite have provided a valuable open source application that users and companies can use to test and evaluate their specific demands regarding serverless computing. Since all is packaged with Docker it is easy to setup and use and no big time commitment needs to be made to conduct some tests.
With the gathered results this thesis has established that serverless computing can definitely be useful and even powerful depending on the scenario, cloud and runtime.
Unfortunately, there is no common framework or standard used by the cloud providers at the expense of the user or customer. This application tries to smooth the way for the user in testing and analyzing the results in order that later deciding on the most suitable serverless platform is more easy to do.

5.2 Future Work

Although this benchmark suite covers the most important aspects it can be developed and be improved further. Apart from general improvements and optimizations more runtimes respectively languages and cloud providers could be supported, more different tests could be implemented and provided. In addition, a plotting integration (e.g. with R) would be useful as Grafana does not provide much exporting and plotting capabilities and is rather focused on live monitoring.
There could be a test regarding continuous deployment and inspect the behaviour of the platform. Besides, a special focus could lay on deploying and testing Docker images because that might be an important aspect and trend in the future, considering there are no runtime restrictions of the provider. Several providers already offer to deploy just a Docker image (AWS, Azure, IBM) and Google has its service Cloud Run which has become generally available on November 14, 2019 [Gooh] and is a fully managed, serverless container platform.
Moreover, the load test should probably be carried out with different function execution times to get a better understanding of the possible connection between scaling and execution time, if there is any at all. Also, a real world application test would be beneficial and demonstrate the usability of serverless computing.

Appendix A

Flow charts

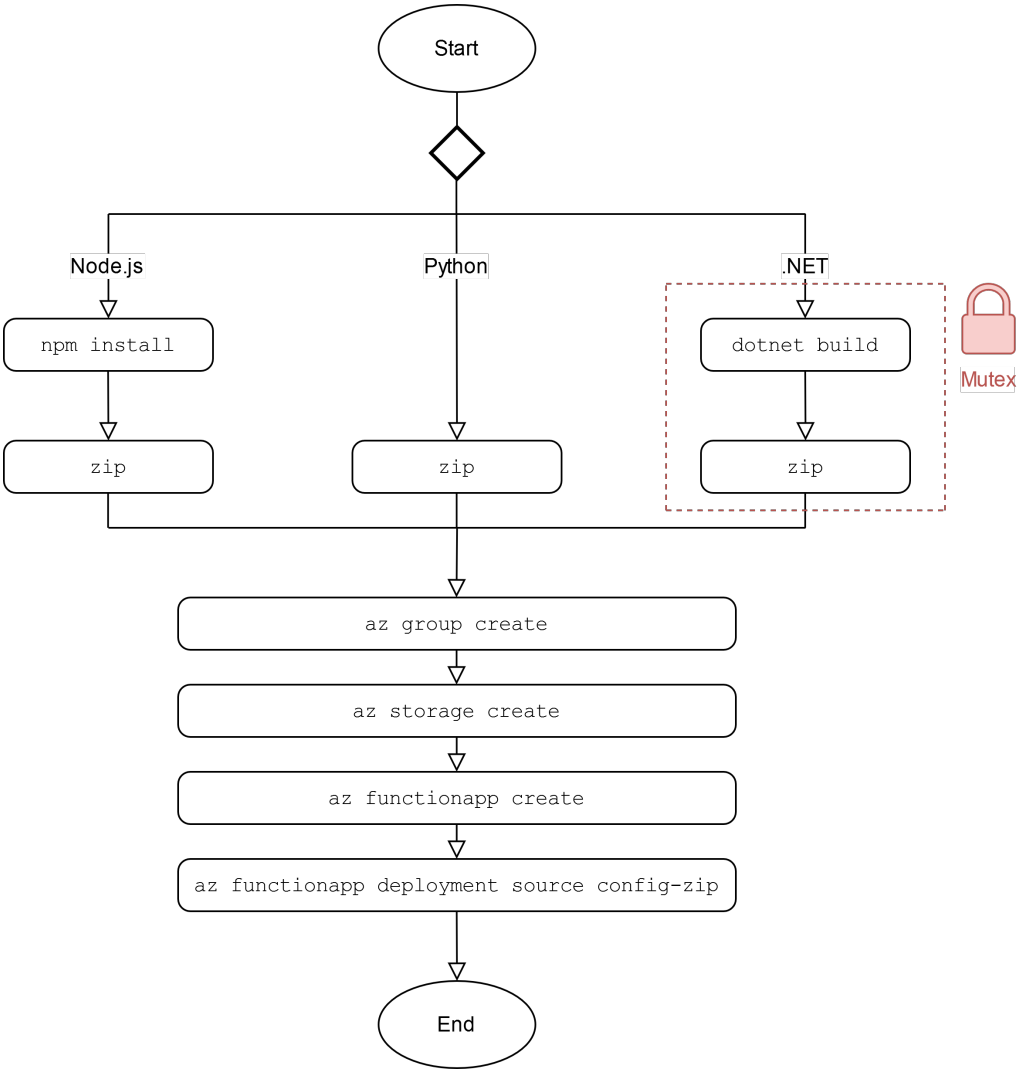

Figure A.1: Azure deployment flow chart
Source: illustration by author

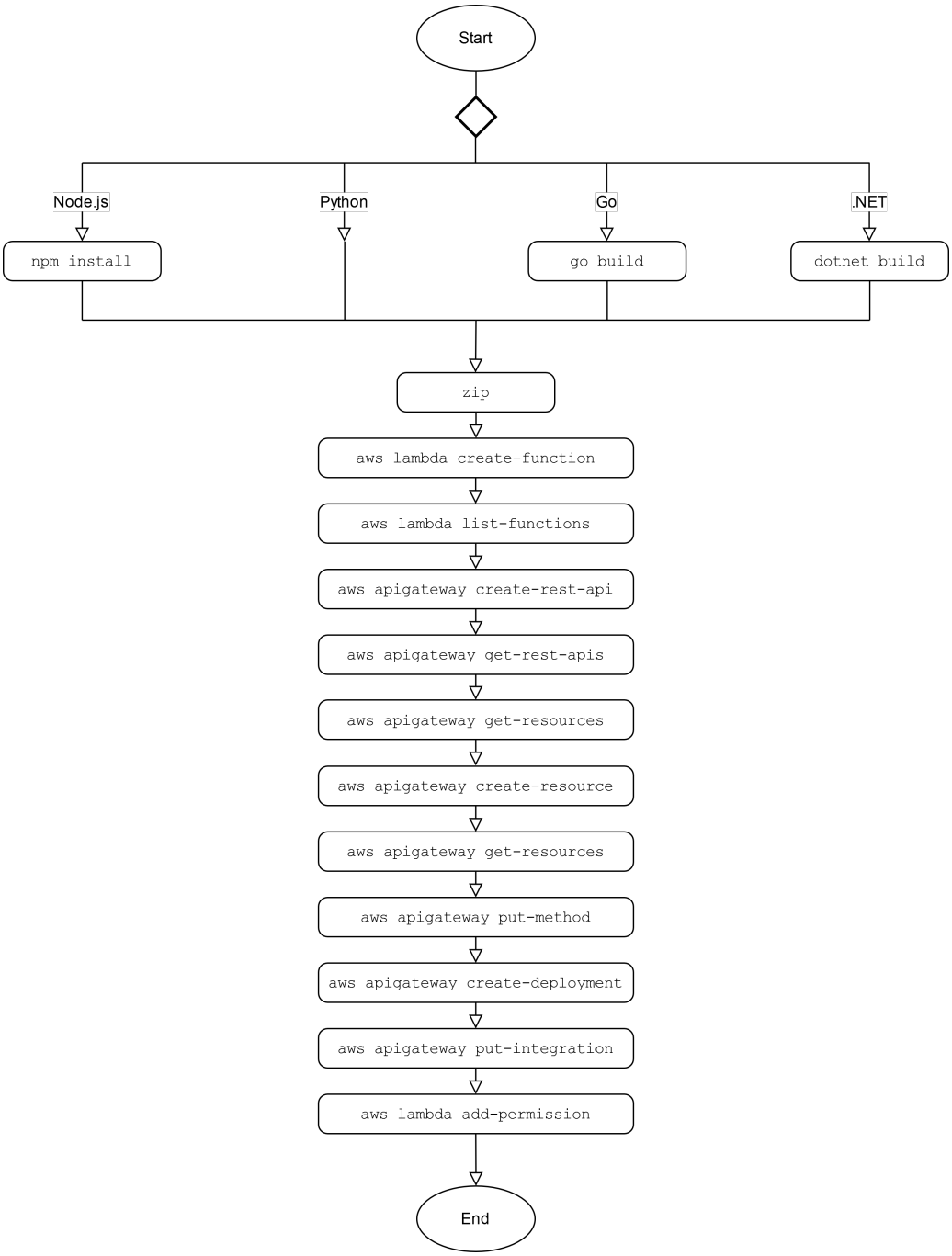

Figure A.2: AWS deployment flow chart
Source: illustration by author

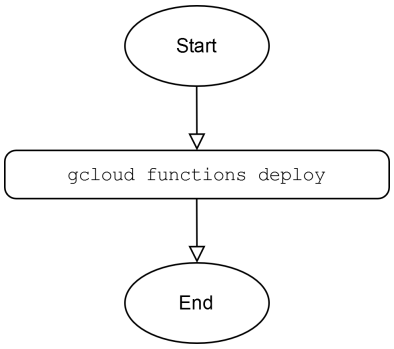

Figure A.3: Google deployment flow chart
Source: illustration by author

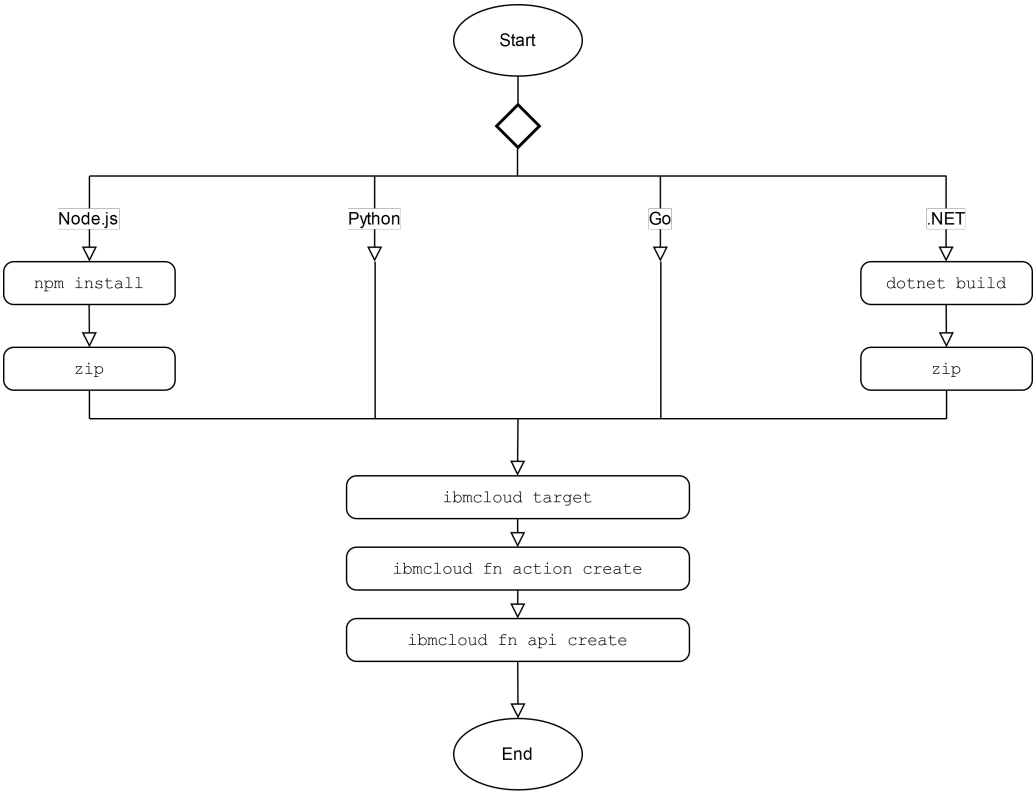

Figure A.4: IBM deployment flow chart
Source: illustration by author

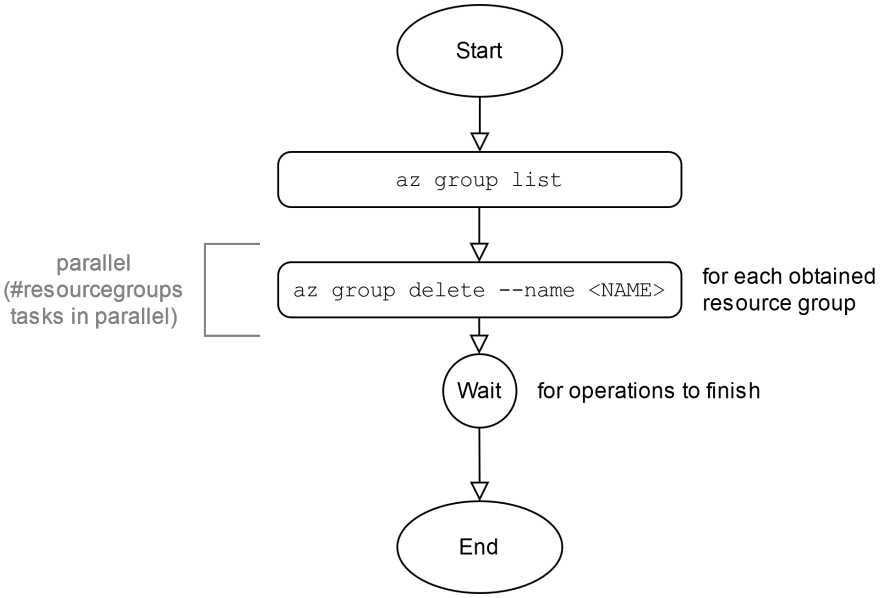

Figure A.5: Azure cleanup flow chart
Source: illustration by author

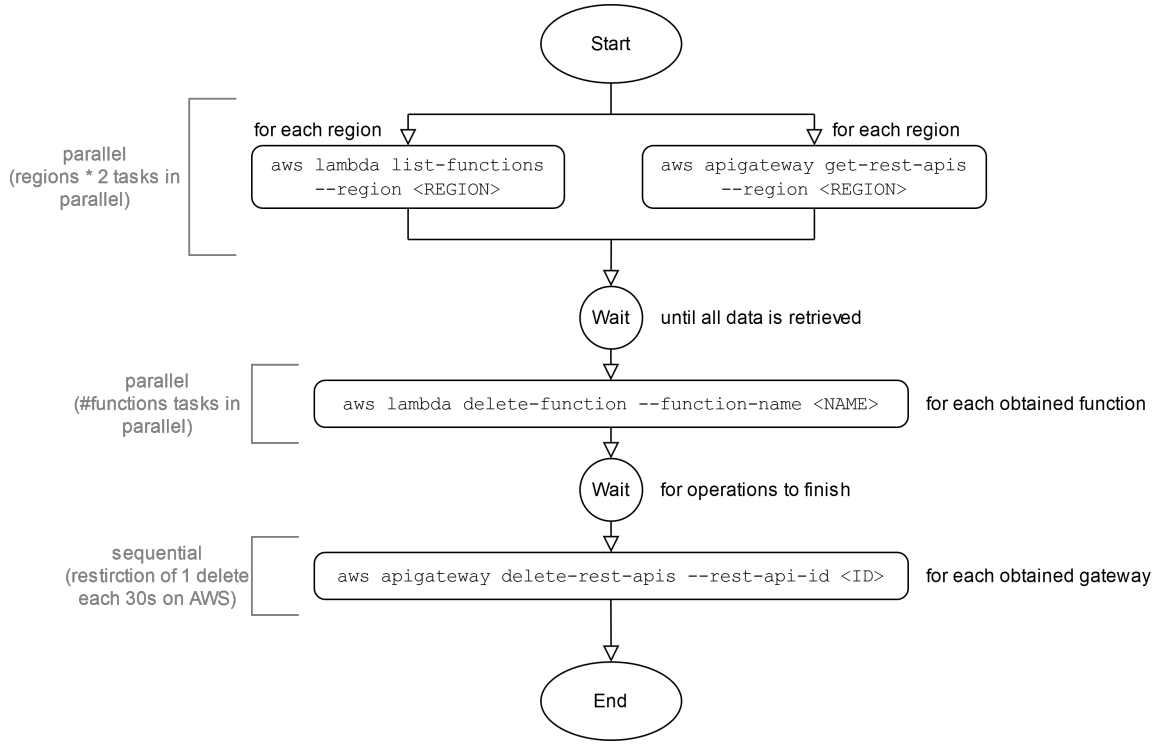

Figure A.6: AWS cleanup flow chart
Source: illustration by author

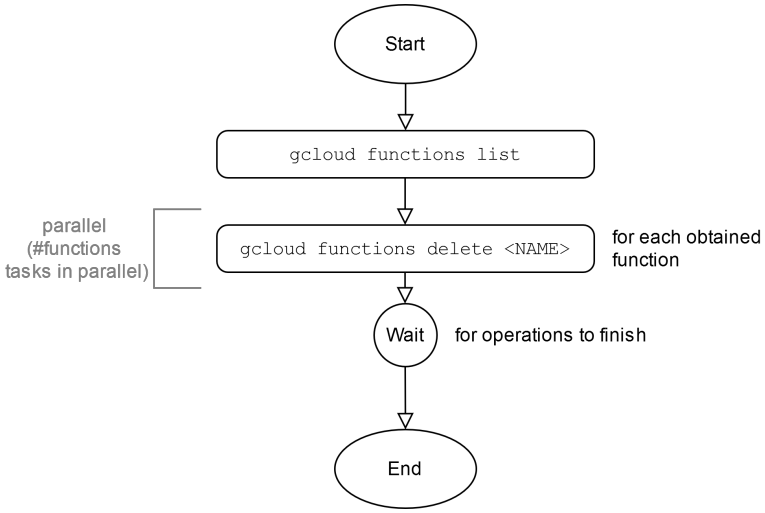

Figure A.7: Google cleanup flow chart
Source: illustration by author

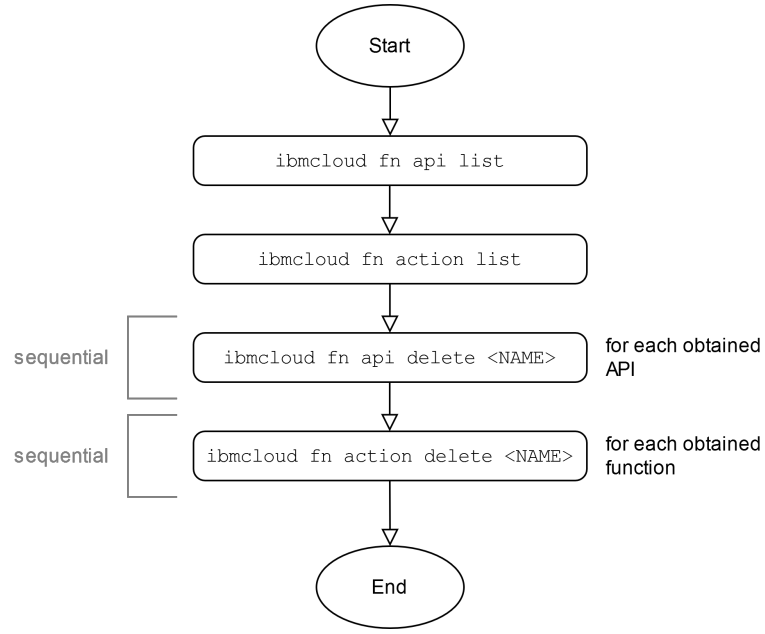

Figure A.8: IBM cleanup flow chart
Source: illustration by author

Appendix B

Results

Cloud	Region	Dist. (km)	Avg.	Min.	Max.	Median	Q_1	Q_3
AWS	eu-central-1	365	80	51	328	77	67	88
IBM	eu-de	365	205	97	2162	160	149	189
AWS	eu-west-3	435	128	69	415	125	104	144
Google	europe-west1	471	111	52	329	98	76	133
Azure	westeurope	630	190	109	758	181	159	208
AWS	eu-west-2	747	144	76	378	146	110	168
Azure	uksouth	747	208	111	774	194	157	234
Google	europe-west2	747	182	53	571	186	134	223
IBM	eu-gb	747	227	124	1121	221	202	233
AWS	eu-west-1	1290	170	87	377	176	161	190
Azure	northeurope	1290	245	129	716	220	197	273
AWS	eu-north-1	1544	159	82	602	154	135	174
AWS	me-south-1	4423	499	189	938	505	486	534
AWS	ca-central-1	5477	356	160	653	397	193	421
IBM	us-east	6600	478	173	1023	486	467	505
AWS	ap-south-1	6622	315	151	522	366	186	390
AWS	us-east-1	6721	311	146	481	355	185	371
Azure	eastus	6721	511	331	1239	475	458	542
Azure	eastus2	6721	547	356	1168	504	480	552
Google	us-east4	6721	291	130	812	187	153	247
Google	us-east1	7220	496	146	1241	308	176	795
AWS	us-east-2	7225	364	155	484	388	376	398
Google	us-central1	7439	222	152	762	203	177	254
Azure	southcentralus	8371	600	434	1115	613	585	633
IBM	us-south	8371	618	220	1308	611	590	660
Azure	westus2	8508	728	517	1846	718	704	760
AWS	us-west-2	8690	515	223	1048	585	569	599
AWS	ap-northeast-2	8865	809	323	1628	873	856	893
AWS	us-west-1	9284	468	209	1138	541	246	556
Azure	westus	9284	703	668	1668	677	673	686
AWS	ap-east-1	9402	640	246	1280	659	646	680
Azure	eastasia	9402	791	586	912	800	789	840
Google	asia-east2	9402	1771	318	2726	2006	1980	2050
AWS	sa-east-1	9531	674	270	1223	726	708	749
AWS	ap-northeast-1	9669	808	298	1818	797	775	818
Google	asia-northeast1	9669	1502	284	1902	1691	1667	1721
IBM	jp-tok	9669	1287	1145	2095	1240	1212	1272
AWS	ap-southeast-1	10381	549	222	1169	555	544	569
Azure	southeastasia	10381	735	571	1190	719	701	742
AWS	ap-southeast-2	16663	893	356	1313	956	948	972
Azure	australiaeast	16663	1304	1251	1889	1269	1258	1308

Table B.1: Latency test details, all numbers in milliseconds except for distance, distances calculated with https://www.distance.to/

Cloud	Runtime	Avg.	Min.	Max.	Median	Q_1	Q_3
AWS	Node.js	352	275	845	298	285	315
AWS	Python	246	224	276	247	231	257
AWS	Go	267	223	357	264	239	277
AWS	.NET	1689	1539	1798	1701	1639	1739
Azure	Node.js	2511	2275	2795	2482	2396	2630
Azure	Python	3019	2325	4517	3026	2537	3138
Azure	.NET	3637	2507	4937	3747	3131	3850
Azure (Windows)	Node.js	4086	3226	4974	4046	3782	4358
Azure (Windows)	.NET	1864	1256	3419	1635	1441	1897
Google	Node.js	1933	1559	2294	1897	1691	2215
Google	Python	2694	2160	3847	2482	2410	2784
Google	Go	1613	1178	2067	1564	1487	1745
IBM	Node.js	826	779	925	819	796	850
IBM	Python	646	599	675	650	633	665
IBM	Go	788	733	906	771	761	799
IBM	.NET	1670	1583	1829	1672	1631	1685

Table B.2: Cold start test details, all numbers in milliseconds

Cloud	Runtime	Memory	Avg.	Min.	Max.	Median	Q_1	Q_3
AWS	Node.js	128	17031	16454	18377	16816	16676	17101
AWS	Node.js	256	8697	8199	9298	8552	8312	9143
AWS	Node.js	512	4367	4233	4569	4319	4276	4462
AWS	Node.js	1024	2078	2045	2118	2077	2065	2088
AWS	Node.js	2048	1135	1113	1195	1132	1125	1139
AWS	Python	128	7903	7809	8040	7903	7870	7927
AWS	Python	256	3935	3890	4017	3928	3915	3942
AWS	Python	512	1948	1915	1973	1947	1942	1955
AWS	Python	1024	1000	981	1020	1000	996	1005
AWS	Python	2048	564	550	591	566	555	571
AWS	Go	128	934	848	1004	940	912	960
AWS	Go	256	454	412	487	460	442	465
AWS	Go	512	221	196	237	220	216	228
AWS	Go	1024	103	91	112	104	99	107
AWS	Go	2048	63	62	70	63	63	63
AWS	.NET	128	669	631	716	666	658	679
AWS	.NET	256	330	306	364	331	323	336
AWS	.NET	512	160	146	180	159	155	163
AWS	.NET	1024	80	74	88	79	78	83
AWS	.NET	2048	51	50	64	50	50	50
Azure	Node.js	1536	2157	1993	2552	2134	2064	2227
Azure	Python	1536	1267	1167	1764	1227	1205	1299
Azure	.NET	1536	92	80	127	89	87	95
AzureWindows	Node.js	1536	6157	5190	12717	5748	5309	6426
AzureWindows	.NET	1536	253	231	400	250	245	254
Google	Node.js	128	16535	2822	25818	15900	14923	18535
Google	Node.js	256	8302	2715	12968	8201	7446	9296
Google	Node.js	512	3484	2088	5476	3399	3307	3532
Google	Node.js	1024	1886	1685	2368	1838	1812	1922
Google	Node.js	2048	1751	1346	5634	1572	1512	1662
Google	Python	128	7653	1229	11697	7598	7006	8502
Google	Python	256	3119	2893	3344	3111	3052	3198
Google	Python	512	1502	1285	1726	1512	1435	1544
Google	Python	1024	874	743	1307	865	824	901
Google	Python	2048	767	534	1933	716	669	827
Google	Go	128	965	725	1287	954	901	1004
Google	Go	256	406	304	600	406	396	414
Google	Go	512	217	160	370	216	194	228
Google	Go	1024	117	77	279	113	98	123
Google	Go	2048	119	75	405	103	83	129
IBM	Node.js	128	2284	1110	9135	1613	1156	1745
IBM	Node.js	256	1608	1523	1944	1546	1538	1613
IBM	Node.js	512	1693	1525	1777	1703	1686	1721
IBM	Node.js	1024	1600	1523	1892	1556	1543	1690
IBM	Node.js	2048	1694	1520	1746	1696	1684	1708
IBM	Python	128	669	584	697	676	674	680
IBM	Python	256	591	576	663	588	585	592
IBM	Python	512	593	577	676	589	586	594
IBM	Python	1024	604	593	652	601	598	607
IBM	Python	2048	603	568	727	597	592	600
IBM	Go	128	93	57	114	95	94	96
IBM	Go	256	60	56	74	59	58	60
IBM	Go	512	60	56	68	60	59	61
IBM	Go	1024	60	56	72	60	59	61
IBM	Go	2048	60	56	67	60	59	61
IBM	.NET	128	52	46	90	47	47	48
IBM	.NET	256	85	83	95	84	84	85
IBM	.NET	512	83	81	90	83	82	85
IBM	.NET	1024	83	81	94	82	82	83
IBM	.NET	2048	82	81	93	82	82	82

Table B.3: General test details, all numbers in milliseconds except memory in MB

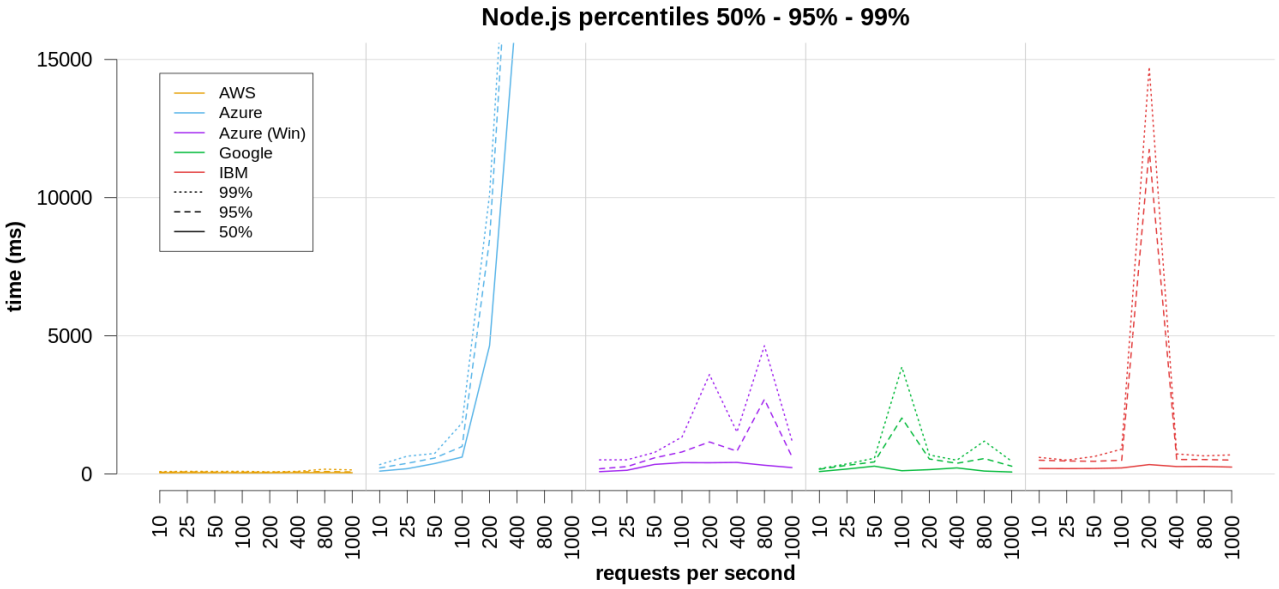

Figure B.1: Latency percentiles (50%, 95%, 99%) Node.js load test
Source: illustration by author

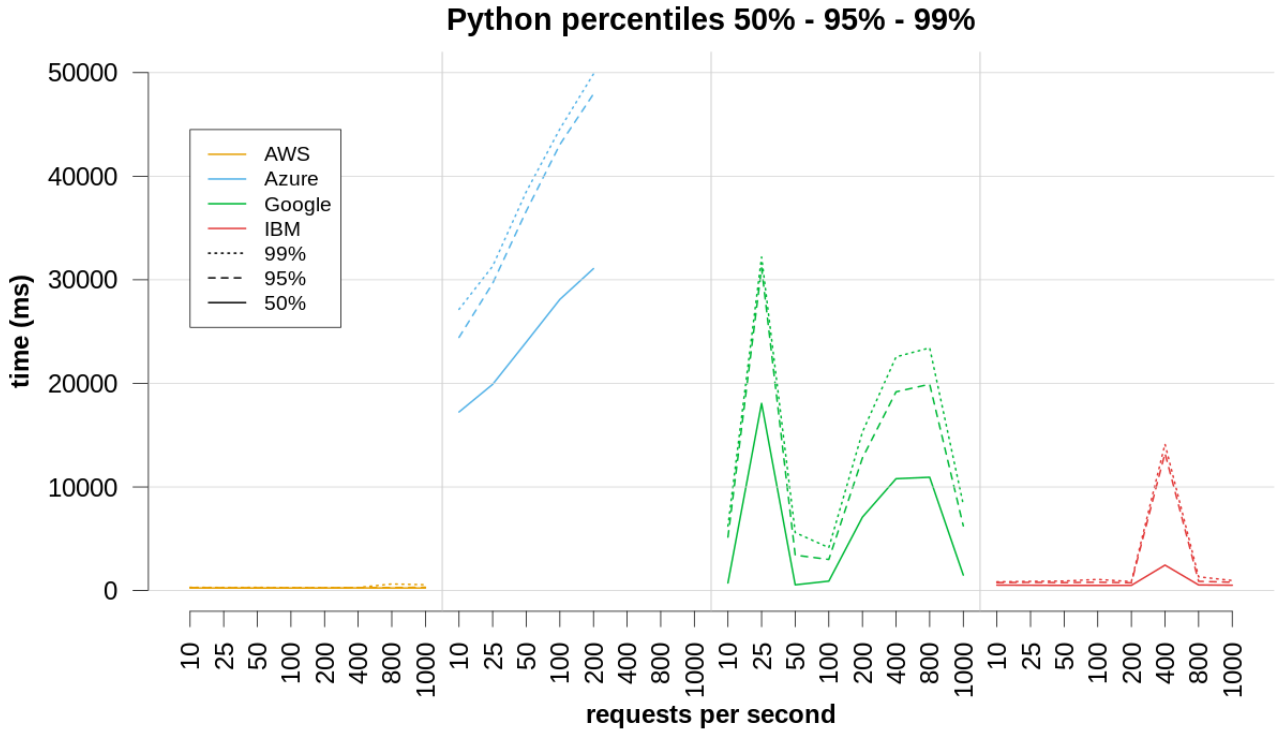

Figure B.2: Latency percentiles (50%, 95%, 99%) Python load test
Source: illustration by author

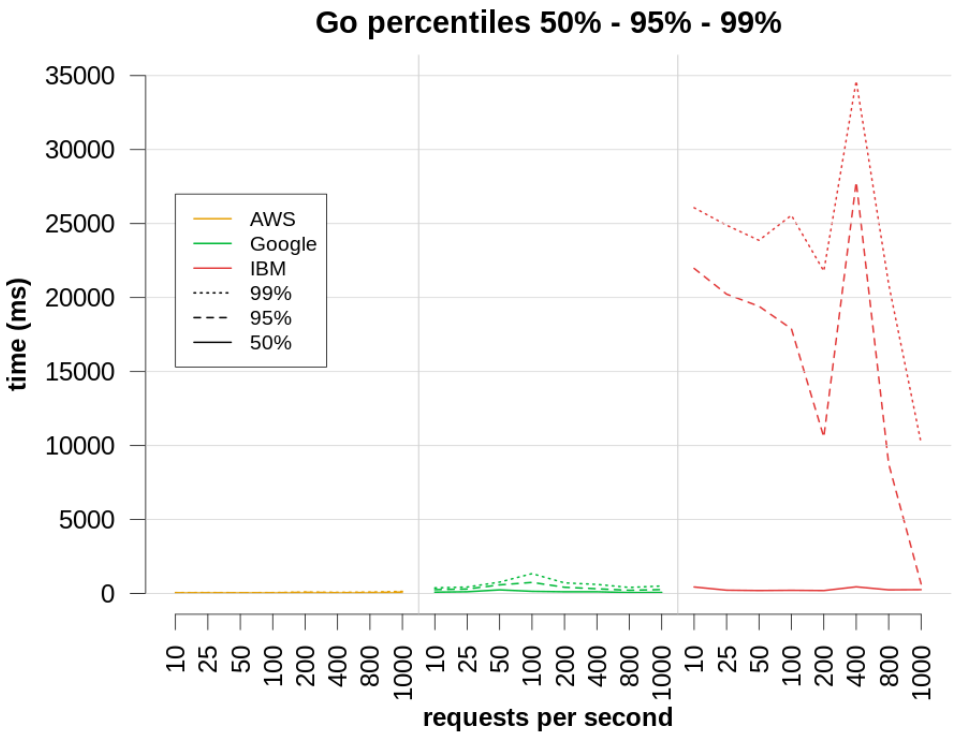

Figure B.3: Latency percentiles (50%, 95%, 99%) Go load test
Source: illustration by author

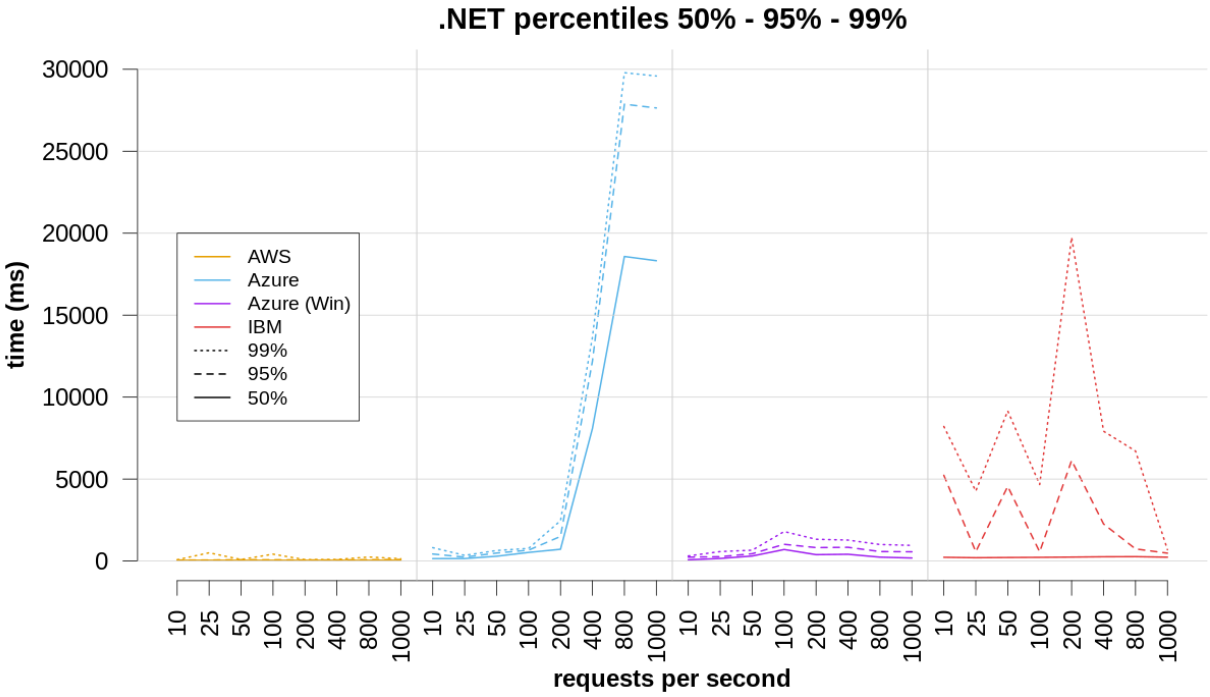

Figure B.4: Latency percentiles (50%, 95%, 99%) .NET load test
Source: illustration by author

Runtime	RPS (goal)	RPS (actual)	Avg.	50%	95%	99%
Node.js	10	10	48ms	49ms	63ms	88ms
Node.js	25	25	51ms	49ms	79ms	98ms
Node.js	50	49	49ms	48ms	69ms	95ms
Node.js	100	98	49ms	47ms	68ms	98ms
Node.js	200	197	47ms	47ms	65ms	80ms
Node.js	400	393	53ms	51ms	77ms	97ms
Node.js	800	786	58ms	51ms	92ms	175ms
Node.js	1000	979	53ms	48ms	77ms	149ms
Python	10	10	255ms	252ms	293ms	315ms
Python	25	25	249ms	247ms	273ms	300ms
Python	50	50	248ms	246ms	269ms	310ms
Python	100	98	249ms	247ms	267ms	289ms
Python	200	197	249ms	247ms	268ms	288ms
Python	400	393	257ms	257ms	285ms	299ms
Python	800	785	263ms	251ms	298ms	638ms
Python	1000	978	264ms	251ms	323ms	567ms
Go	10	10	40ms	42ms	53ms	67ms
Go	25	25	39ms	41ms	51ms	71ms
Go	50	49	40ms	41ms	52ms	61ms
Go	100	98	42ms	41ms	55ms	68ms
Go	200	197	47ms	44ms	89ms	106ms
Go	400	393	44ms	44ms	61ms	85ms
Go	800	786	46ms	43ms	73ms	104ms
Go	1000	977	56ms	47ms	103ms	151ms
.NET	10	10	46ms	46ms	61ms	81ms
.NET	25	25	50ms	45ms	63ms	504ms
.NET	50	49	50ms	47ms	82ms	94ms
.NET	100	98	56ms	47ms	79ms	422ms
.NET	200	197	49ms	47ms	66ms	87ms
.NET	400	393	49ms	46ms	68ms	99ms
.NET	800	786	53ms	47ms	84ms	246ms
.NET	1000	978	54ms	49ms	94ms	143ms

Table B.4: AWS load test details. 50% meaning that 50% of the requests could been served in that time or less. Same applies for 95% and 99%.

OS	Runtime	RPS (goal)	RPS (actual)	Avg.	50%	95%	99%
Linux	Node.js	10	10	123ms	105ms	221ms	340ms
Linux	Node.js	25	25	205ms	193ms	388ms	646ms
Linux	Node.js	50	49	375ms	379ms	579ms	744ms
Linux	Node.js	100	98	640ms	614ms	997ms	1850ms
Linux	Node.js	200	179	4755ms	4650ms	8495ms	10130ms
Linux	Node.js	400	227	16505ms	17190ms	24642ms	26100ms
Linux	Node.js	800	278	23934ms	24480ms	37027ms	38700ms
Linux	Node.js	1000	327	24218ms	24840ms	37388ms	39290ms
Linux	Python	10	6	16498ms	17240ms	24444ms	27150ms
Linux	Python	25	12	19444ms	19910ms	29655ms	31310ms
Linux	Python	50	18	23427ms	23990ms	36635ms	38500ms
Linux	Python	100	25	27261ms	28110ms	43057ms	44560ms
Linux	Python	200	30	30448ms	31080ms	47940ms	49870ms
Linux	Python	400	15	9365ms	8640ms	23167ms	28480ms
Linux	Python	800	14	6949ms	5520ms	19382ms	23950ms
Linux	Python	1000	15	7327ms	6170ms	19447ms	24670ms
Linux	.NET	10	10	204ms	150ms	430ms	817ms
Linux	.NET	25	25	157ms	152ms	241ms	347ms
Linux	.NET	50	49	310ms	294ms	489ms	644ms
Linux	.NET	100	98	515ms	527ms	665ms	772ms
Linux	.NET	200	197	783ms	720ms	1502ms	2460ms
Linux	.NET	400	324	8152ms	8090ms	12198ms	13660ms
Linux	.NET	800	407	18140ms	18580ms	27869ms	29790ms
Linux	.NET	1000	512	18012ms	18320ms	27640ms	29590ms
Windows	Node.js	10	10	102ms	86ms	190ms	513ms
Windows	Node.js	25	25	150ms	136ms	269ms	515ms
Windows	Node.js	50	49	335ms	348ms	585ms	780ms
Windows	Node.js	100	98	413ms	413ms	799ms	1340ms
Windows	Node.js	200	197	515ms	409ms	1161ms	3600ms
Windows	Node.js	400	393	436ms	420ms	829ms	1520ms
Windows	Node.js	800	785	672ms	318ms	2703ms	4640ms
Windows	Node.js	1000	979	282ms	234ms	630ms	1210ms
Windows	.NET	10	10	99ms	77ms	238ms	311ms
Windows	.NET	25	25	158ms	158ms	266ms	579ms
Windows	.NET	50	49	277ms	307ms	447ms	662ms
Windows	.NET	100	98	671ms	705ms	1022ms	1800ms
Windows	.NET	200	197	423ms	391ms	822ms	1330ms
Windows	.NET	400	393	433ms	414ms	839ms	1280ms
Windows	.NET	800	786	274ms	238ms	580ms	1010ms
Windows	.NET	1000	979	230ms	187ms	566ms	956ms

Table B.5: Azure load test details. 50% meaning that 50% of the requests could been served in that time or less. Same applies for 95% and 99%.

Runtime	RPS (goal)	RPS (actual)	Avg.	50%	95%	99%
Node.js	10	10	99ms	90ms	131ms	193ms
Node.js	25	25	183ms	181ms	242ms	367ms
Node.js	50	50	265ms	285ms	339ms	572ms
Node.js	100	99	362ms	120ms	208ms	3870ms
Node.js	200	197	211ms	159ms	265ms	688ms
Node.js	400	393	228ms	222ms	291ms	500ms
Node.js	800	786	181ms	108ms	182ms	1190ms
Node.js	1000	978	113ms	73ms	147ms	460ms
Python	10	10	1356ms	740ms	1240ms	6230ms
Python	25	12	18448ms	18070ms	24560ms	32190ms
Python	50	49	915ms	557ms	918ms	5600ms
Python	100	98	1200ms	914ms	1590ms	4150ms
Python	200	168	7315ms	7090ms	9630ms	15300ms
Python	400	297	10871ms	10800ms	14300ms	22560ms
Python	800	575	11144ms	10940ms	14680ms	23430ms
Python	1000	976	2246ms	1500ms	3360ms	8310ms
Go	10	10	113ms	97ms	122ms	386ms
Go	25	25	138ms	125ms	164ms	442ms
Go	50	49	271ms	252ms	350ms	776ms
Go	100	98	241ms	154ms	265ms	1350ms
Go	200	197	172ms	123ms	212ms	724ms
Go	400	393	151ms	124ms	178ms	620ms
Go	800	786	98ms	70ms	124ms	416ms
Go	1000	979	131ms	78ms	140ms	512ms

Table B.6: Google load test details. 50% meaning that 50% of the requests could been served in that time or less. Same applies for 95% and 99%.

Runtime	RPS (goal)	RPS (actual)	Avg.	50%	95%	99%
Node.js	10	10	289ms	201ms	500ms	602ms
Node.js	25	25	259ms	197ms	473ms	504ms
Node.js	50	50	252ms	201ms	462ms	635ms
Node.js	100	98	280ms	221ms	499ms	900ms
Node.js	200	197	2707ms	342ms	11747ms	14730ms
Node.js	400	393	313ms	271ms	526ms	727ms
Node.js	800	784	293ms	275ms	524ms	657ms
Node.js	1000	978	277ms	252ms	505ms	694ms
Python	10	10	565ms	531ms	774ms	845ms
Python	25	25	563ms	518ms	798ms	891ms
Python	50	49	536ms	503ms	769ms	921ms
Python	100	98	545ms	502ms	809ms	1080ms
Python	200	197	527ms	505ms	751ms	866ms
Python	400	382	4640ms	2460ms	13255ms	14140ms
Python	800	781	593ms	545ms	894ms	1330ms
Python	1000	977	547ms	518ms	809ms	973ms
Go	10	10	5296ms	447ms	21971ms	26070ms
Go	25	21	3600ms	230ms	20234ms	24890ms
Go	50	50	3373ms	205ms	19415ms	23870ms
Go	100	98	2731ms	221ms	17891ms	25560ms
Go	200	198	1331ms	204ms	10568ms	21790ms
Go	400	390	7027ms	457ms	27820ms	34640ms
Go	800	777	1222ms	253ms	8774ms	20910ms
Go	1000	979	498ms	266ms	654ms	10180ms
.NET	10	10	848ms	230ms	5231ms	8210ms
.NET	25	25	380ms	204ms	571ms	4270ms
.NET	50	49	717ms	217ms	4555ms	9160ms
.NET	100	94	391ms	227ms	559ms	4670ms
.NET	200	196	1121ms	242ms	6136ms	19740ms
.NET	400	393	616ms	262ms	2263ms	7920ms
.NET	800	785	488ms	269ms	741ms	6710ms
.NET	1000	978	263ms	236ms	482ms	691ms

Table B.7: IBM load test details. 50% meaning that 50% of the requests could been served in that time or less. Same applies for 95% and 99%.

Appendix C

Miscellaneous

Listing C.1: Google Cloud Functions - Example content of /proc/cpuinfo

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 85
model name      : unknown
stepping        : unknown
cpu MHz         : 2000.162
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           :
bogomips        : 2000.16
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

Listing C.2: IBM Cloud Functions - Example content of /proc/cpuinfo

```
processor       : [0..3]
vendor_id       : GenuineIntel
cpu family      : 6
model           : 63
model name      : Intel(R) Xeon(R) CPU E5-2683 v3 @ 2.00GHz
stepping        : 2
microcode       : 0x43
cpu MHz         : 2000.056
cache size      : 35840 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
```

```
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           :
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs
bogomips        : 4000.11
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

List of Figures

Fig. 1.1: On premises - IaaS - PaaS - SaaS . . . 3
Fig. 1.2: High-level serverless FaaS platform architecture . . . 4

Fig. 2.1: Cloud Infrastructure Services Market Share . . . 7

Fig. 3.1: Benchmark Suite Architecture . . . 16
Fig. 3.2: Web GUI - Deploy/Delete Tests . . . 19
Fig. 3.3: Test results in Grafana . . . 20
Fig. 3.4: Benchmark results in Grafana . . . 21

Fig. 4.1: Latency test scatter plot . . . 23
Fig. 4.2: Cold and warm start on Azure . . . 24
Fig. 4.3: Cold start latency box plot . . . 24
Fig. 4.4: Scatter plot of execution times of the CPU factors test in Python . . . 25
Fig. 4.5: Load test average latency . . . 27
Fig. 4.6: Azure Linux .NET Live Metrics Stream during 1000 RPS . . . 28
Fig. 4.7: Google Go active instances during load test . . . 29

Fig. A.1: Azure deployment flow chart . . . 38
Fig. A.2: AWS deployment flow chart . . . 39
Fig. A.3: Google deployment flow chart . . . 40
Fig. A.4: IBM deployment flow chart . . . 40
Fig. A.5: Azure cleanup flow chart . . . 41
Fig. A.6: AWS cleanup flow chart . . . 41
Fig. A.7: Google cleanup flow chart . . . 42
Fig. A.8: IBM cleanup flow chart . . . 42

Fig. B.1: Latency percentiles (50%, 95%, 99%) Node.js load test . . . 46
Fig. B.2: Latency percentiles (50%, 95%, 99%) Python load test . . . 46
Fig. B.3: Latency percentiles (50%, 95%, 99%) Go load test . . . 47
Fig. B.4: Latency percentiles (50%, 95%, 99%) .NET load test . . . 47

List of Tables

Tab. 2.1: Google Cloud Functions - Possible memory allocation and corresponding CPU frequency . 9
Tab. 2.2: Supported runtimes in serverless computing 10
Tab. 2.3: Supported Node.js runtimes 10
Tab. 2.4: Supported Python runtimes 11
Tab. 2.5: Supported Go runtimes 11
Tab. 2.6: Supported .NET Core runtimes 12

Tab. 3.1: Docker images 18

Tab. 4.1: Load tests that achieved less than 90% of RPS 29
Tab. 4.2: Serverless functions pricing 30
Tab. 4.3: Serverless functions free tier 30
Tab. 4.4: Pricing example regarding test results 32
Tab. 4.5: Summary and comparison of services 35

Tab. B.1: Latency test details 43
Tab. B.2: Cold start test details 44
Tab. B.3: General test details 45
Tab. B.4: AWS load test details 48
Tab. B.5: Azure load test details 49
Tab. B.6: Google load test details 50
Tab. B.7: IBM load test details 51

Glossary

ACU Azure Compute Unit. 8
API Application Programming Interface. 17, 19, 33
AWS Amazon Web Services. ii, 2, 4, 7, 8, 10, 12, 17, 19, 23–28, 30–34
CEO Chief Executive Officer. 6
CLI Command Line Interface. 2, 9, 17, 18, 32–34
CPU Central Processing Unit. 7–9, 13, 14, 20, 25, 26, 31, 33, 36, 54
CSS Cascading Style Sheets. 10
EC2 Elastic Compute Cloud. 2
FaaS Function as a Service. ii, 1, 3, 4, 35
GB Gigabyte. 8, 22
GHz Gigahertz. 7–9
GUI Graphical User Interface. 17
HTML Hypertext Markup Language. 10
HTTP Hypertext Transfer Protocol. 4, 12, 17, 27, 28
IaaS Infrastructure as a Service. 2
IBM International Business Machines Corporation. ii, 4, 6, 7, 9, 10, 12, 23–26, 28, 30, 31, 33, 34
JSON JavaScript Object Notation. 12
MB Megabyte. 7, 9, 19, 22, 23, 25, 26, 30, 31, 45
MHz Megahertz. 9
NIST National Institute of Standards and Technology. 2, 3
NPM Node Package Manager. 10
OS Operating System. 2, 9, 10
PaaS Platform as a Service. 2, 4
RPS Requests per second. 28
SaaS Software as a Service. 2, 4
vCPU virtual Central Processing Unit. 7, 8
VM Virtual Machine. 2, 4, 8, 35

References

[Ali19] Alibaba Cloud: *What Is IaaS?* https://www.alibabacloud.com/de/knowledge/what-is-iaas. Version: 2019. – Last accessed: 18.12.2019

[Apa] Apache: *Apache OpenWhisk*. https://openwhisk.apache.org/. – Last accessed: 05.11.2019

[AWSa] AWS, Amazon Web Services: *AWS Lambda*. https://aws.amazon.com/lambda/. – Last accessed: 22.10.2019

[AWSb] AWS, Amazon Web Services: *AWS Lambda Function Configuration*. https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html. – Last accessed: 22.10.2019

[AWSc] AWS, Amazon Web Services: *AWS Lambda Function Scaling*. https://docs.aws.amazon.com/lambda/latest/dg/scaling.html. – Last accessed: 22.10.2019

[AWSd] AWS, Amazon Web Services: *AWS Lambda Pricing*. https://aws.amazon.com/lambda/pricing/. – Last accessed: 22.01.2020

[AWSe] AWS, Amazon Web Services: *AWS Lambda Releases*. https://docs.aws.amazon.com/lambda/latest/dg/lambda-releases.html. – Last accessed: 22.10.2019

[AWSf] AWS, Amazon Web Services: *AWS Lambda Runtimes*. https://docs.aws.amazon.com/lambda/latest/dg/lambda-runtimes.html. – Last accessed: 22.10.2019

[AWSg] AWS, Amazon Web Services: *AWS Region Table*. https://aws.amazon.com/about-aws/global-infrastructure/regional-product-services/. – Last accessed: 10.12.2019

[AWS19] AWS, Amazon Web Services: *Serverless*. https://aws.amazon.com/serverless/. Version: 2019. – Last accessed: 18.12.2019

[BCC+17] Baldini, Ioana ; Castro, Paul ; Chang, Kerry ; Cheng, Perry ; Fink, Stephen ; Ishakian, Vatche ; Mitchell, Nick ; Muthusamy, Vinod ; Rabbah, Rodric ; Slominski, Aleksander ; Suter, Philippe: Serverless Computing: Current Trends and Open Problems. In: Chaudhary, Sanjay (Hrsg.) ; Somani, Gaurav (Hrsg.) ; Buyya, Rajkumar (Hrsg.): *Research Advances in Cloud Computing*. Singapore : Springer Singapore, 2017. – ISBN 978–981–10–5026–8, 1–20

[Bun19] Bundesamt für Sicherheit in der Informationstechnik: *Cloud Comptuing Grundlagen*. https://www.bsi.bund.de/EN/Topics/CloudComputing/Basics/Basics_node.html. Version: 2019. – Last accessed: 22.10.2019

[CIMS19] Castro, Paul ; Ishakian, Vatche ; Muthusamy, Vinod ; Slominski, Aleksander: The Rise of Serverless Computing. In: *Commun. ACM* 62 (2019), November, Nr. 12, 44–54. http://dx.doi.org/10.1145/3368454. – DOI 10.1145/3368454. – ISSN 0001–0782

[CSS19] Conger, Kate ; Sanger, David E. ; Shane, Scott: *Microsoft Wins Pentagon's $ 10 Billion JEDI Contract, Thwarting Amazon*. https://www.nytimes.com/2019/10/25/technology/dod-jedi-contract.html. Version: 25.10.2019. – Last accessed: 26.11.2019

[EIST17] Eyk, Erwin van ; Iosup, Alexandru ; Seif, Simon ; Thömmes, Markus: The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures. In: *Proceedings of the 2Nd International Workshop on Serverless Computing*. New York, NY, USA : ACM, 2017 (WoSC '17). – ISBN 978–1–4503–5434–9, 1–4

[FGZ+18] Figiela, Kamil ; Gajek, Adam ; Zima, Adam ; Obrok, Beata ; Malawski, Maciej: Performance evaluation of heterogeneous cloud functions. In: *Concurrency and Computation: Practice and Experience* 30 (2018), Nr. 23, e4792. http://dx.doi.org/10.1002/cpe.4792. – DOI 10.1002/cpe.4792. – e4792 cpe.4792

[Go19a] Go: *Documentation*. https://golang.org/doc/. Version: 2019. – Last accessed: 10.12.2019

[Go19b] Go: *The Go Project*. https://golang.org/project/. Version: 2019. – Last accessed: 10.12.2019

[Gooa] Google: *Cloud Functions*. https://cloud.google.com/functions. – Last accessed: 29.10.2019

[Goob] Google: *Cloud Functions - Cloud Functions Execution Environment*. https://cloud.google.com/functions/docs/concepts/exec. – Last accessed: 26.11.2019

[Gooc] Google: *Cloud Functions - Pricing*. https://cloud.google.com/functions/pricing#compute_time. – Last accessed: 30.10.2019

[Good] Google: *Cloud Functions - Quotas*. https://cloud.google.com/functions/quotas. – Last accessed: 30.10.2019

[Gooe] Google: *Cloud Functions - Release Notes*. https://cloud.google.com/functions/docs/release-notes. – Last accessed: 30.10.2019

[Goof] Google: *Cloud Functions - Writing Cloud Functions*. https://cloud.google.com/functions/docs/writing. – Last accessed: 30.10.2019

[Goog] Google: *Cloud Functions Locations*. https://cloud.google.com/functions/docs/locations. – Last accessed: 10.12.2019

[Gooh] Google: *Cloud Run (fully managed) release notes*. https://cloud.google.com/run/docs/release-notes. – Last accessed: 07.02.2020

[Goo19] Google: *Serverless computing*. https://cloud.google.com/serverless/. Version: 2019. – Last accessed: 18.12.2019

[GZC+19] Gan, Yu ; Zhang, Yanqi ; Cheng, Dailun ; Shetty, Ankitha ; Rathi, Priyal ; Katarki, Nayan ; Bruno, Ariana ; Hu, Justin ; Ritchken, Brian ; Jackson, Brendon ; Hu, Kelvin ; Pancholi, Meghna ; He, Yuan ; Clancy, Brett ; Colen, Chris ; Wen, Fukang ; Leung, Catherine ; Wang, Siyuan ; Zaruvinsky, Leon ; Espinosa, Mateo ; Lin, Rick ; Liu, Zhongling ; Padilla, Jake ; Delimitrou, Christina: An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA : ACM, 2019 (ASPLOS '19). – ISBN 978–1–4503–6240–5, 3–18

[IBMa] IBM: *Cloud Foundry*. https://www.ibm.com/cloud/cloud-foundry. – Last accessed: 07.02.2020

[IBMb] IBM: *IBM Cloud Functions*. https://cloud.ibm.com/functions. – Last accessed: 30.10.2019

[IBMc] IBM: *IBM Cloud Functions - Pricing*. https://cloud.ibm.com/functions/learn/pricing. – Last accessed: 22.01.2020

[IBM19a] IBM: *IBM Cloud Functions - Runtimes*. https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-runtimes. Version: 04.09.2019. – Last accessed: 05.11.2019

[IBM19b] IBM: *IBM Cloud Functions - Regions*. https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-cloudfunctions_regions. Version: 12.07.2019. – Last accessed: 10.12.2019

[IBM19c] IBM: *IBM Cloud Functions - System details and limits*. https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-limits. Version: 18.09.2019. – Last accessed: 05.11.2019

[IBM19d] IBM: *Cloud computing: A complete guide*. https://www.ibm.com/cloud/learn/cloud-computing. Version: 2019. – Last accessed: 22.10.2019

[IBM19e] IBM: *Serverless computing*. https://www.ibm.com/cloud/learn/serverless. Version: 2019. – Last accessed: 18.12.2019

[Kir] Kiriaty, Yochay: *Announcing general availability of Azure Functions*. https://azure.microsoft.com/en-us/blog/announcing-general-availability-of-azure-functions/. – Last accessed: 29.10.2019

[KNJ18] Kuntsevich, Aleksandr ; Nasirifard, Pezhman ; Jacobsen, Hans-Arno: A Distributed Analysis and Benchmarking Framework for Apache OpenWhisk Serverless Platform. In: *Proceedings of the 19th International Middleware Conference (Posters)*. New York, NY, USA : ACM, 2018 (Middleware '18). – ISBN 978–1–4503–6109–5, 3–4

[Lan19] Lander, Richard: *Announcing .NET Core 3.1*. https://devblogs.microsoft.com/dotnet/announcing-net-core-3-1/. Version: 2019. – Last accessed: 11.12.2019

[LSF18] Lee, Hyungro ; Satyam, Kumar ; Fox, Geoffrey: Evaluation of Production Serverless Computing Environments, 2018

[MFGZ18] Malawski, Maciej ; Figiela, Kamil ; Gajek, Adam ; Zima, Adam: Benchmarking Heterogeneous Cloud Functions. In: Heras, Dora B. (Hrsg.) ; Bougé, Luc (Hrsg.) ; Mencagli, Gabriele (Hrsg.) ; Jeannot, Emmanuel (Hrsg.) ; Sakellariou, Rizos (Hrsg.) ; Badia, Rosa M. (Hrsg.) ; Barbosa, Jorge G. (Hrsg.) ; Ricci, Laura (Hrsg.) ; Scott, Stephen L. (Hrsg.) ; Lankes, Stefan (Hrsg.) ; Weidendorfer, Josef (Hrsg.): *Euro-Par 2017: Parallel Processing Workshops*. Cham : Springer International Publishing, 2018. – ISBN 978–3–319–75178–8, S. 415–426

[MG11] Mell, Peter M. ; Grance, Timothy: *SP 800-145. The NIST Definition of Cloud Computing*. Gaithersburg, MD, United States : National Institute of Standards & Technology, 2011 https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf. – Last accessed: 22.10.2019

[Mica] Microsoft: *Azure Functions*. https://azure.microsoft.com/en-us/services/functions/. – Last accessed: 07.02.2020

[Micb] Microsoft: *Azure Functions pricing*. https://azure.microsoft.com/en-us/pricing/details/functions/. – Last accessed: 22.01.2020

[Micc] Microsoft: *Azure Functions runtime versions overview*. https://docs.microsoft.com/en-us/azure/azure-functions/functions-versions. – Last accessed: 15.01.2020

[Micd] Microsoft: *Azure Functions scale and hosting*. https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale. – Last accessed: 29.10.2019

[Mice] Microsoft: *General purpose virtual machine sizes - Av2-series*. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes-general#av2-series. – Last accessed: 29.10.2019

[Micf] Microsoft: *Products available by region*. https://azure.microsoft.com/en-us/global-infrastructure/services/?products=functions&regions=all. – Last accessed: 10.12.2019

[Micg] Microsoft: *Supported languages in Azure Functions*. https://docs.microsoft.com/en-us/azure/azure-functions/supported-languages. – Last accessed: 29.10.2019

[Mich] Microsoft: *What is .NET?* https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet. – Last accessed: 11.12.2019

[Mic16] Microsoft: *.NET Core 1.0 Releases*. https://github.com/dotnet/core/tree/master/release-notes/1.0. Version: 2016. – Last accessed: 11.12.2019

[Mic19a] Microsoft: *.NET Core Guide*. https://docs.microsoft.com/en-us/dotnet/core/. Version: 2019. – Last accessed: 11.12.2019

[Mic19b] Microsoft: *Serverless computing*. https://azure.microsoft.com/en-us/overview/serverless-computing/. Version: 2019. – Last accessed: 18.12.2019

[Noda] Node.js Foundation: *Node.js*. https://nodejs.org/. – Last accessed: 07.11.2019

[Nodb] Node.js Foundation: *Node.js Releases*. https://nodejs.org/en/about/releases/. – Last accessed: 26.11.2019

[Pic19] Pichai, Sundar: *Unleashing digital opportunities in Europe*. https://www.blog.google/around-the-globe/google-europe/unleashing-digital-opportunities-europe/. Version: 20.09.2019. – Last accessed: 22.10.2019

[Pyta] Python Software Foundation: *Python - About*. https://www.python.org/about/. – Last accessed: 11.12.2019

[Pytb] Python Software Foundation: *Quotes about Python*. https://www.python.org/about/quotes/. – Last accessed: 11.12.2019

[Syn19] Synergy Research Group: *Cloud Service Spending Still Growing Almost 40of it Won by Amazon & Microsoft*. https://www.srgresearch.com/articles/cloud-service-spending-still-growing-almost-40-year-half-it-won-amazon-microsoft. Version: 26.07.2019. – Last accessed: 30.10.2019

[Tea15] Team, Microsoft Windows S.: *Microsoft Loves Linux*. https://cloudblogs.microsoft.com/windowsserver/2015/05/06/microsoft-loves-linux/. Version: 2015. – Last accessed: 11.12.2019

[Ten19] Tene, Gil: *wrk2 - a HTTP benchmarking tool based mostly on wrk*. https://github.com/giltene/wrk2. Version: 2019. – Last accessed: 04.02.2020

[Tre18] Tresness, Colby: *Understanding serverless cold start*. https://azure.microsoft.com/en-us/blog/understanding-serverless-cold-start/. Version: 07.02.2018. – Last accessed: 22.01.2020

[US 19] US Department of Defense: *Contracts For Oct. 25, 2019 - WASHINGTON HEADQUARTERS SERVICE*. https://www.defense.gov/Newsroom/Contracts/Contract/Article/1999639/. Version: 25.10.2019. – Last accessed: 26.11.2019

[w3s] w3schools: *Python Introduction*. https://www.w3schools.com/python/python_intro.asp. – Last accessed: 11.12.2019

[Wik19] Wikipedia: *JavaScript*. https://de.wikipedia.org/w/index.php?title=JavaScript&oldid=193743331. Version: 2019. – Last accessed: 26.11.2019

[WLZ+18] Wang, Liang ; Li, Mengyuan ; Zhang, Yinqian ; Ristenpart, Thomas ; Swift, Michael: Peeking Behind the Curtains of Serverless Platforms. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA : USENIX Association, 2018. – ISBN 978–1–931971–44–7, 133–146