university of groningen

faculty of science and engineering

Master's Thesis

University of Groningen

Faculty of Science and Engineering

Hybrid serverless and virtual machine deployment model for cost minimization of cloud applications

Author:
Timon Back

Primary Supervisor:
Dr.
Vasilios Andrikopoulos

Secondary Supervisor:
Prof. Dr.
Alexander Lazovik

August 29, 2018

Abstract

The Function-as-a-Service (FaaS) subtype of serverless cloud computing provides the means for abstracting away from servers on which developed software is meant to be executed. It essentially offers an event-driven and scalable environment in which billing is based on the actual resource consumption and not on the provisioning of resources.

As research is limited in this field, this thesis provides an overview of the state-of-the-art along with a performance benchmark, which reveals significant differences between the cloud service providers and a field still under heavy development.

Moreover, a generic decision model for a hybrid deployment strategy using FaaS and traditional Virtual Machines in the public cloud model is proposed, which is implemented in the form of a simulator. The simulator assists in finding the Pareto optimal hybrid deployment strategy for cost minimization of any cloud application. In addition to low load scenarios, this strategy is especially efficient for bursty load scenarios.

Keywords

- Benchmark
- Cloud computing
- Cost-Management
- Deployment Model
- Function-as-a-Service (FaaS)
- Pareto Optimality
- Scalable applications
- Serverless
- Simulator

Acknowledgements

I would like to express my gratitude to everyone who have made it possible for me to complete this thesis.

First of all, I would like to thank my supervisors Vasilios Andrikopoulos and Alexander Lazovik for their professional and accurate support they have given me during the research.

Further I want to thank my fellow students for the talks and motivation throughout the execution and writing of this work.

List of Figures

2.1 Abstractions of the service models in cloud computing 6
2.2 Popularity of the term *serverless* on Google Trends 7
2.3 The serverless model in the context of service models 10
2.4 Serverless platform architecture 10
2.5 Cost model of serverless computing 15
2.6 Comparison of the cost of the VM-based and serverless execution model 16

3.1 Measured successful durations of PI2048 across all cloud service providers 26
3.2 Measured successful durations of FFT in different memory configuration 27
3.3 Cumulative execution times of FFT in different memory configurations using only the successful executions ($k \in [8.192; 131.072]$) . 28
3.4 Ranking of the cloud service providers of the cloud functions . . . 29
3.5 Measured successful durations of S2048 across all cloud service providers 29
3.6 Measured successful durations of PI2048 across all cloud service providers 31
3.7 Measured successful durations of UF2048 across all cloud service providers 32

4.1 Overview of the decision model 37

5.1 Visualization of a constant and bursty load 49
5.2 Load distribution from different artificial load patterns 50
5.3 Load distribution sampled (n=3.600) from the normal distribution using different sigma values 51
5.4 Simplified architecture of the implemented simulator 52
5.5 Overview of the load patterns implemented in the simulator . . . 55
5.6 Screenshot of the web-interface of the simulator 56
5.7 Overview of the Pareto front for the different load patterns 61
5.8 Pareto front for loads following the normal distribution with different sigma values (8x less requests compared to Figure 5.7) . . . 61

A.1 Measured successful durations of S128 across all cloud service provider 73

B.1 Pareto optimal deployment strategies for the constant and squared load pattern with varying altitudes 79
B.2 Pareto optimal deployment strategies for the periodic load patterns with varying altitudes 80

List of Tables

2.1 Allocated CPU cycles per memory configuration in the Google Cloud Functions environment according to Google LLC (2018b) . 14
2.2 Comparison of the major cloud service providers based on the data available at the point of writing 17

3.1 Relative resource requirements of the benchmarked cloud functions 22
3.2 Used parameters for the benchmarked cloud functions 24
3.3 Mean Square Error (MSE) for linear regression to the observed data of S per memory configuration per cloud service provider (taken from Back and Andrikopoulos (2018)) 30
3.4 Successful executions of FFT across all memory configurations per parameter k (taken from Back and Andrikopoulos (2018)) 32
3.5 Measured cumulative duration in seconds of FFT over different memory configurations and cloud service providers 33

5.1 Measured throughput and response time in seconds on AWS EC2 with the *m5.xlarge* instance type 59
5.2 Measured and trend line duration of FFT 59
5.3 Cost comparison of a full FaaS, hybrid and full VM-based deployment solutions 62
5.4 Cost improvement in percentage compared to an all VM-based deployment strategy 63

A.1 Speed-up of the cloud functions on AWS Lambda over different memory configurations 72
A.2 Speed-up of the cloud functions on Google Cloud Functions over different memory configurations 72
A.3 Measured memory consumption of FFT in MB on AWS Lambda using the maximum value 74
A.4 Measured memory consumption of MM in MB on AWS Lambda using the maximum value 74
A.5 Measured memory consumption of PI in MB on AWS Lambda using the maximum value 74
A.6 Measured memory consumption of UF in MB on AWS Lambda using the maximum value 74

B.1 Parameters used in the simulation 78

Glossary

Apache OpenWhisk is an open-source FaaS implementation. The (incubating) Apache project is used by IBM Cloud Functions. Only implementation, which has no limitations on the used programming language due to the use of Docker. Apache OpenWhisk (2018).

AWS (Amazon Web Services) is the largest public cloud service provider according to RightScale (2018) and first major one to offer FaaS.

AWS Lambda is the FaaS offering in the AWS cloud. Amazon Web Services, Inc. (2018a).

benchmark is "a standardized problem or test that serves as a basis for evaluation or comparison (as of computer system performance)" Merriam-Webster (2018a). Commonly used to measure and compare the performance of different systems and at different points in time.

BTU (Billing Time Unit) is the minimal possible billing interval of an instance. While the BTU of a VM is relatively large, the BTU of FaaS lies in the area of milliseconds.

cloud is the metaphorical place where cloud computing happens. Typically, this is at the premise (data center) of a cloud service provider.

cloud computing is a paradigm, where computer hardware is rented offside for the time of usage. Resources are used on-demand over the internet in a pay-as-you-go model. The cloud service providers (like AWS, Google Cloud Functions, IBM Cloud Functions, Microsoft Azure Functions) take care of managing the hardware without interruptions (availability).

cloud service provider is the operator of a cloud computing environment. The big cloud service provider of FaaS in public clouds are AWS Lambda, Google Cloud Functions, IBM Cloud Functions and Microsoft Azure Functions are described in Section 2.3.

Docker is a containerization tool using the Linux kernel. Docker allows the

encapsulation of processes in an own environment without starting up a complete operating system within a VM, which makes its usage light on resources (Docker, Inc, 2018). Its adoption is around 50% of cloud companies according to RightScale (2018).

FaaS (Function-as-a-Service) is a specific runtime implementation of the serverless cloud computing execution model.

Google Cloud Functions is the FaaS offering in the Google Cloud. Google LLC (2018a).

IBM Cloud Functions is the FaaS offering in the IBM Cloud (IBM Bluemix before). Uses internally Apache OpenWhisk. IBM (2018).

Microsoft Azure Functions is the FaaS offering in the Microsoft Azure cloud. Microsoft Corporation (2018).

Pareto optimal describes a solution for which any reallocations of resources worsens at least one decision criterion. While a reallocation can improve one or more decision criterions, at least is decreased. Named after Vilfredo Pareto. Common concept in economics and game theory (Pardalos et al., 2008).

public cloud is a cloud that is offered by a cloud service provider to anyone, who is willing to purchase the service. In opposite to a private cloud hosted usually on-site of a company, a public cloud scales easily from the perspective of the consumer and maintenance is done by the cloud service provider. A special case is the hybrid cloud, where a private and public cloud get combined. (Mell and Grance, 2011).

serverless (or serverless computing) is a cloud execution model, where typically a cloud service provider manages the resources. The consumer is billed in short BTUs for the consumed resources in a pay-as-you-go model instead of pre-allocating and provisioning of machines. An extensive background is provided in Section 2.2.

simulator is one that simulates. “A device that enables the operator to reproduce or represent under test conditions phenomena likely to occur in actual performance” Merriam-Webster (2018b). Using a model, a situation can be reproduced and optimized.

VM stands for virtual machine. It is an emulation of a computer system. On non-dedicated virtual machines, the emulated hardware is shared between multiple virtual machines to gain better total utilization.

Contents

Abstract iii

List of Figures vii

List of Tables ix

Glossary xi

1 Introduction 1
- 1.1 Context & Scope . . . 1
- 1.2 Problem Definition . . . 2
- 1.3 Research Method . . . 3
- 1.4 Outline . . . 4

2 Background 5
- 2.1 Cloud Computing & Service Models . . . 5
- 2.2 Serverless Computing . . . 6
  - 2.2.1 Characteristics . . . 7
  - 2.2.2 Cloud Functions . . . 9
  - 2.2.3 Function-as-a-Service (FaaS) . . . 9
  - 2.2.4 Use-Cases . . . 11
- 2.3 Cloud Service Providers . . . 12
  - 2.3.1 AWS Lambda . . . 13
  - 2.3.2 Google Cloud Functions . . . 13
  - 2.3.3 IBM Cloud Functions . . . 14
  - 2.3.4 Microsoft Azure Functions . . . 14
  - 2.3.5 Cost Model . . . 14
  - 2.3.6 Summary . . . 17

3 Benchmark 19
- 3.1 Related Work . . . 19
- 3.2 Design . . . 20
  - 3.2.1 Hypotheses . . . 20
  - 3.2.2 Functions . . . 21
- 3.3 Setup & Execution . . . 23
- 3.4 Results . . . 24
  - 3.4.1 Hypotheses . . . 25

3.4.2 Memory Consumption . . . 31
3.4.3 Cloud Function Termination . . . 31
3.4.4 Cost . . . 33
3.5 Limitations . . . 35
3.6 Lessons Learned . . . 35
3.7 Conclusion . . . 36

4 Decision Model **37**
4.1 Related work . . . 38
4.2 Assumptions . . . 38
4.3 Parameters . . . 40
4.3.1 Load . . . 40
4.3.2 Instance Types & Pricing . . . 41
4.4 Strategy . . . 41
4.4.1 Steps . . . 42
4.4.2 Limitations . . . 44
4.5 Summary . . . 44

5 Simulator **45**
5.1 Requirements . . . 46
5.2 Design Decisions . . . 48
5.2.1 Auto-scaling & Request Queuing . . . 48
5.2.2 Load Patterns & Characteristics . . . 49
5.2.3 Technology Selection . . . 51
5.3 Implementation . . . 51
5.3.1 Architecture . . . 52
5.3.2 Decision Model . . . 52
5.3.3 Features . . . 53
5.4 Decision Model Configuration . . . 57
5.4.1 Obtaining the Configuration Values . . . 58
5.5 Results . . . 60
5.5.1 Constant and Squared Load . . . 60
5.5.2 Periodic Load . . . 60
5.5.3 Bursty Load . . . 61
5.5.4 Cost-Efficiency . . . 62
5.6 Conclusion . . . 63

6 Conclusion **65**
6.1 Summary . . . 65
6.2 Future Work . . . 66

A Benchmark **69**
A.1 Execution Script . . . 69
A.2 Serverless Wrapper Code . . . 71
A.3 BH-2 Speed-ups between Memory Configurations . . . 72
A.4 BH-4: The billed duration matches the execution time . . . 73
A.5 Memory Consumption . . . 74

B Simulator **75**

B.1 Standalone Version of the Cloud Functions 75
B.2 Code for Finding the Trend Line 77
B.3 Evaluation 78
B.3.1 Parameters 78
B.3.2 Constant and Squared Load 79
B.3.3 Periodic Load 80

Bibliography **81**

Declaration of Authorship **87**

Chapter 1

Introduction

The wide adoption of the cloud and cloud computing in past years has changed the way digital services are being offered. Concepts like microservices and containerization have innovated the distribution and execution of complete software stacks into portable packages by simplifying the installation and removing platform specific configuration. Serverless computing is the next step to go (Buyya et al., 2017).

20% of the respondents of the annual RightScale State of the Cloud Report (RightScale, 2018) already use serverless products and another 40% are testing or looking into it with a growth rate of 75% between 2017 and 2018. According to Barga (2017), in 2017 Thomas Reuters processed already 4.000 requests per second, Expedia 1,2 billion requests each month and Vevo is able to handle spikes of 80x the normal traffic each using the serverless implementation AWS Lambda. Alone the revenue of AWS has seen a steady increase, with $17,5 billion in 2017 making cloud computing a billion dollar business (Statista, 2018).

Still, managing cost expenses and the lack of expertise are part of the big challenges faced by companies according to RightScale (2018).

1.1 Context & Scope

Serverless computing is a new paradigm in cloud computing, where the cloud service provider manages the resource allocation and pricing is based on actual usage as explained in more detail in Section 2.2.

Recently, serverless computing started moving into the focus of the industry as the concept is promising and major public cloud service providers have been pushing their runtime models to the market.

The event-driven nature of serverless computing using stateless cloud functions enables instant scaling of cloud applications. The resource management lies in the responsibility of the cloud service provider, so that the consumer does not have to worry about the infrastructure anymore. Moreover, resources are billed based on the actual usage compared to provisioned resources irrespectively of their utilization in the traditional cloud computing billing model of e.g. VMs.

Therefore, serverless computing is seen as a chance for substantial cost reduction of cloud applications. Although the cost model explained in Section 2.3.5 is more complex, besides low usage applications, especially bursty and compute-intensive application benefit from the serverless computing (Baldini et al., 2017).

Still, academic research in this field is limited, while the industry has started adopting and integrating the Function-as-a-Service (FaaS) subtype of serverless computing. FaaS is one way to implement the serverless computing execution model and is covered in more detail in Section 2.2.3. This work aims at providing a more complete picture of the opportunities of FaaS from both, the academic and industrial perspective.

The work by van Eyk et al. (2017) tries to clarify the concepts of the serverless paradigm. While seeing the need for a "performance- and cost-metric", the task was left as future work.

Outside of the scope of this work lies the task of building serverless applications or refactoring previous as monolith, service oriented architecture (SOA) or microservice written applications to fit the serverless model. The work of Spillner (2017b) provides an automatic tool for transforming Python code to AWS Lambda and Spillner and Dorodko (2017) offer a tool for the transformation of Java code. Belloum (2017) extends an existing toolkit to enable cloud function deployments to AWS Lambda. Moreover, the initial migration to a cloud service provider with maximum gain in utility is discussed in Gómez Sáez et al. (2018).

1.2 Problem Definition

The goal of this work is to answer the following research questions. They are split into two parts, as the second question builds on top of the first question.

What is the performance of public Function-as-a-Service solutions?

As mentioned before, the field of serverless computing is still under development and so are the FaaS solutions implementing the serverless model. While the major cloud service providers are offering their FaaS implementations to the public, not sufficient academic comparison of the performance and cost relationship between them exists.

Only one of the major cloud service provider releases information about performance guarantees, however on an abstract level.

Additionally, the multiple layers of abstraction and visualization of the underlying infrastructure hides the execution environment and impacts the performance. Roberts and Chapin (2017) notice that even between identically configured cloud functions, the performance can vary drastically.

What is the optimal deployment strategy to handle load cost-efficiently using a combination of FaaS- and VM-based instances?

The literature (see for example Albuquerque Jr et al. (2017), Baldini et al. (2017)) claims that FaaS is best suited to handle unpredictable load due to its possibility to scale easily, while predictable load can be best handled with VM-based solutions. First of all, this claim needs to be verified. Secondly, no decision support based on load or load characteristics is offered.

Further: Are there cloud applications that should move fully to serverless computing or are not suited for the serverless model? In which cases, is a hybrid deployment strategy best in terms of cost efficiency and by which cost?

What are the characteristics of a load distribution to be best served by either of these two cloud computing service models? Or is the mixture of both the most cost-efficient solution?

These questions were asked before, but previous research only answered for certain scenarios or case studies e.g. in Albuquerque Jr et al. (2017), Jonas et al. (2017), Villamizar et al. (2016). A generic approach is missing at the moment.

1.3 Research Method

The research question determines the research method. Therefore, each research question has its own way of approach as presented in the next paragraphs.

What is the performance of public Function-as-a-Service solutions?

To evaluate the performance of system, typically a standardized benchmark is used (see Li et al. (2013)). Chapter 3 introduces a new, simple benchmark, which includes multiple cloud functions with different resource consumptions. To answer the research question and create a long-term beneficial benchmark, multiple aspects are considered.

First, for the benchmark commonly known and used algorithms are selected. Those algorithms are representative for many algorithm classes and typical for cloud scenarios.

Second, any benchmark must execute automatically and the results must be reproducible. Impact of external factors is reduced to a minimum.

Third, the gathered results of the selected cloud service provider are compared and analyzed. This step requires the analysis of log files as during the benchmark any interference is avoided. From this data, information about performance, cost and other relations are gained.

What is the optimal deployment strategy to handle load cost-efficiently using a combination of FaaS- and VM-based instances?

The second research question is more complex as it involves more steps to reach a solution. The result from the previous research question is foundation as the performance and cost comparison of FaaS and VM is an essential aspect.

For selecting an optimal deployment strategy, first multiple solutions from the solution space resolving the problem are created and then ranked. The metric of ranking is cost-efficiency. This process is wrapped into a formal decision model. For an optimal decision model also the relevant parameters need to be identified.

Furthermore, the theoretical decision model is implemented in a decision support tool. After determining the parameter values for the decision model, the tool finds the optimal deployment strategy programmatically.

This is done in form of a simulator discussed further in Chapter 5. Using a simulator has the advantages of testing configurations fast, without the need of deploying and measuring an application.

1.4 Outline

Chapter 2 introduces the concepts of serverless computing as well as other fundamental topics that are used in this work.

In Chapter 3, the first research question is tackled by the design and execution of a benchmark. Results are discussed and lessons are learned.

Using the insights of the developed Benchmark, a theoretical Decision Model in Chapter 4 is proposed in order to answer the second research question. The Simulator implements the previous Decision Model from Chapter 5. The tool assists in finding the Pareto optimal deployment strategy for any cloud application.

Finally, the work and scientific contribution are discussed in conclusion in Chapter 6 with a summary in Section 6.1. An outlook on future work is given in Section 6.2.

Chapter 2

Background

This chapter introduces the foundation of this work with a special focus on the serverless paradigm and the offerings of the selected cloud service providers.

2.1 Cloud Computing & Service Models

Cloud computing is defined by Mell and Grance (2011) as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction".

Essential characteristics are (a) the on-demand self-service, where a consumer access resources without human-interaction, (b) the broad network access for accessing the resources, (c) resource pooling, where multiple physical resources are combined, although the location is unknown or abstracted to an availability zone, country, state, data center, (d) elasticity for dynamically scaling up and down of resources rapidly with seemingly unlimited capacity, (e) measured service, that monitors the resource usage for both the cloud service provider and consumer. (Mell and Grance, 2011)

Additionally, multiple service models exist in the cloud, which are shown in Figure 2.1. Their difference is the level of abstraction of the used infrastructure and the operational concerns. It can be seen as a trade-off between control and convenience. Infrastructure-as-a-Service has the greatest freedom in configuration, while in Software-as-a-Service the complete cloud infrastructure is managed by the cloud service provider to offer maximum convenience to the consumer. (Badger et al., 2011, Mell and Grance, 2011)

The consumer has most control over the operational concerns with Infrastructure-as-a-Service (IaaS). On IaaS, the consumer does not control the underlying hardware, but can control the operating system, attached storage and networking.

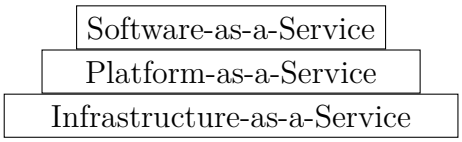

Figure 2.1.: Abstractions of the service models in cloud computing

Thus, any software can be run, but it comes with the responsibility of managing the used instances on a fundamental level. (Badger et al., 2011, Mell and Grance, 2011)

At the other end of the spectrum lies Software-as-a-Service (SaaS). In this service model, the cloud service provider provisions the complete software stack, which consumer access via a thin-client interface like a web-browser. All operational concerns are on the side of the cloud service provider. The consumer has no control over the infrastructure. (Badger et al., 2011, Mell and Grance, 2011)

In between those two models is Platform-as-a-Service (PaaS). The cloud service provider provides the infrastructure including operating system, storage and execution environment of a selected programming language including libraries. The consumer controls the software that is being executed. Maurer et al. (2011) state that if "a consumer has used the facilities of the PaaS cloud to implement and deploy an application, the application essentially is a SaaS deployment". (Badger et al., 2011, Mell and Grance, 2011)

2.2 Serverless Computing

Serverless computing is a new paradigm for deploying cloud applications, where the cloud service provider manages the resource allocation. Its event-driven and scalable environment benefits many application types and is the next step after microservices and containers. The field is gaining momentum over the past years. One indicator is the search popularity of the term *serverless* on Google search as shown in Figure 2.2.

While servers still exist in serverless computing - in contradiction to the name, they are abstracted away from the consumers' perspective. The responsibility of provisioning and scaling of resources moves from the consumer to the cloud service provider (Roberts, 2014). The consumer therefore avoids problems of overprovisioning compared to VM-based approached as the ones discussed in Frey and Hasselbring (2011).

This opens new opportunities. One of the main advantages is the change in operational concerns compared to PaaS. The serverless cloud is fully maintained by the cloud service provider, while reducing the configuration on the side of the consumer. Eivy (2017) notices: "It might save DevOps cost in setting up auto-scale, managing security patches and debugging issues with load balancers at scale".

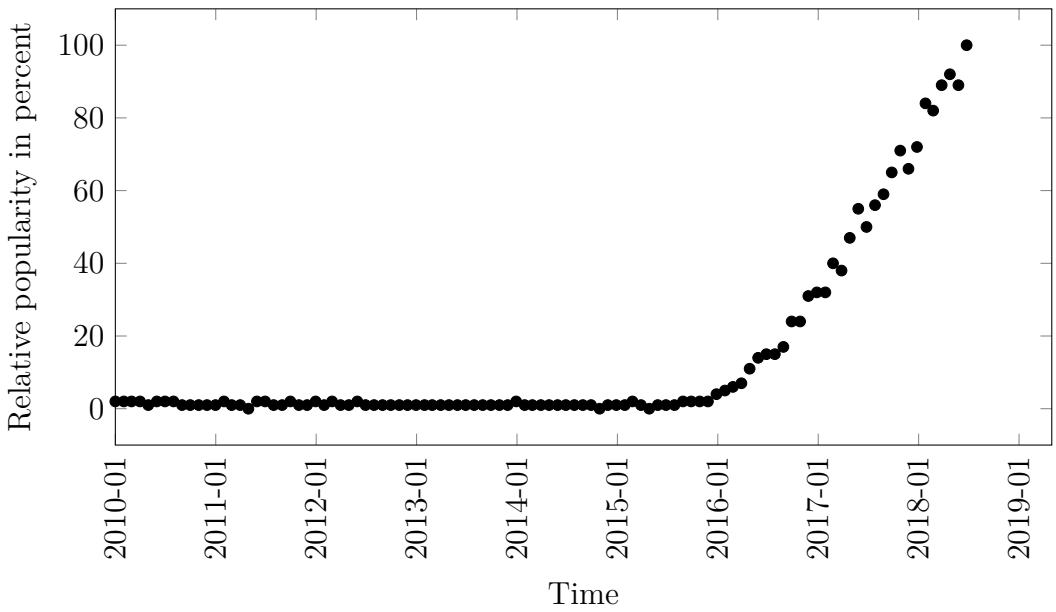

Figure 2.2.: Popularity of the term *serverless* on Google Trends (Google LLC, 2018c)

However, it is important to understand that serverless does not mean no-ops. Although the main task of managing the resources moves towards the cloud service provider, many other tasks like monitoring, support and security audits are still essential. (Roberts and Chapin, 2017)

Moreover, with serverless computing, code deployments become more frequent with a possible shift towards continuous deployments as the code pieces are smaller and replacements or updates are easy and fast to deploy. (Roberts and Chapin, 2017)

However, it comes with the price of modularizing applications. The architecture of the application must support the new paradigm of event-driven invocation as this is one of the key aspects. (Baldini et al., 2017, van Eyk et al., 2017)

The most important change is the cost model, which is covered in Section 2.3.5.

2.2.1 Characteristics

Serverless computing offers multiple advantages compared to traditional VM-based deployment solutions, but also imposes some limitations. These are listed below.

Advantages of Serverless Computing

- Event-driven (Baldini et al., 2017, Müns, 2017)
- Scaling of cloud functions is done automatically (auto-scaling) and handled by the cloud service provider (Baldini et al., 2017, Müns, 2017)

- Focus on the application, not the infrastructure, reduces the lead time Roberts and Chapin (2017)
- No fixed costs, zero cost for idle time, potentially cheaper to run (Baldini et al., 2017, Müns, 2017)
- Better fault tolerance and availability as the cloud service provider manages the resources (Baldini et al., 2017)
- Reduced failure risks as the cloud service providers manages the infrastructure and is more likely to have expertise in that field Roberts and Chapin (2017)
- Automatic security updates (Baldini et al., 2017, Müns, 2017)
- Rich eco system around FaaS by the cloud service provider compared to a local deployment solution
- Update of cloud functions without downtime (Baldini et al., 2017)
- Faster, iterative cloud function deployments through smaller size (Baldini et al., 2017)
- Additional security measures to limit the execution environment and concurrent executions (Baldini et al., 2017)

Disadvantages of Serverless Computing

- Complex applications can be hard to build (Müns, 2017)
- Modularization and FaaSification of cloud functions is difficult (Spillner et al., 2017)
- Not the latest software and programming platforms are available similar to SaaS (Baldini et al., 2017)
- Available memory resources and maximum execution time is limited (Baldini et al., 2017)
- Cost model is more complex and requires thought during development (Baldini et al., 2017, Leitner et al., 2016)
- Cost-riple effect, where one invocation can cause multiple invocations downstream (Kuhlenkamp and Klems, 2017)
- Vendor lock-in to FaaS as well as the eco-system of the cloud service provider - if used Baldini et al. (2017), Roberts and Chapin (2017)
- Possibility higher latency by Cloud functions not immediately ready due to cold start provisioning state (Baldini et al., 2017, Roberts and Chapin, 2017)
- Security is still necessary. Possibility more functions exposed to the outside world and therefore more attack vectors (Baldini et al., 2017)
- Serverless does not mean no-ops. Personnel is still needed for monitoring, support and deployments. (Eivy, 2017, Roberts and Chapin, 2017)
- Local testing is difficult and so is remote debugging (Roberts and Chapin, 2017)

2.2.2 Cloud Functions

van Eyk et al. (2017) define a cloud function as a "small, stateless, on-demand service with a single functional responsibility. The function implements specific business logic, depending on the goal of the application".

The ability to scale without performance loss is made possible by the requirement of being stateless. No state is maintained between any function execution. Instead, information can be retrieved and saved via external services e.g. a (serverless) database or storage system. A context object is accessible to gain limited information about the current execution like the passed in parameters. Still, a cloud function should be idempotent as for the same input, the identical output is expected. (Baldini et al., 2017)

A cloud function is only responsible for a small part of the application and has no knowledge of the overall design or goal of it. It is typically (a) short in duration of a couple seconds to a maximum of minutes, (b) unaware of the context and the reason of invocation and (c) platform-agnostic as only a valid runtime environment is required. (van Eyk et al., 2017)

One of the open questions is still the level of decomposition of cloud functions (FaaSification). The furthest level of decomposition is the creation of a cloud function for every line of code. Spillner et al. (2017) calls that *Deep FaasSifiction*. In *Shallow FaaSifiction* classes or functions are combined into one cloud function. In between lies *Medium FaaSifiction*, which groups function sections into a cloud function. It is important to keep in mind, that a higher decomposition also increases the complexity of the application.

Furthermore, van Eyk et al. (2017) notice that cloud functions can become complex. Therefore, they envision a workflow of composed cloud functions to maintain the flexibility while staying on the serverless abstraction level.

2.2.3 Function-as-a-Service (FaaS)

Function-as-a-Service is a particular implementation of the serverless paradigm to execute cloud functions usually in a public cloud environment. The selected cloud service providers offering FaaS are listed in Section 2.3.

However, it should be noted that Microsoft Azure Functions is the only FaaS implementation that is truly serverless, since it measures the maximum memory consumption dynamically. The other cloud service providers require the developer to specific this parameter during the deployment as will be shown in Section 2.3.5.

In regard to the introduced cloud computing service models in Section 2.1, serverless and the FaaS subtype fits in between of the PaaS and SaaS service model as shown in Figure 2.3. Its emergence is linked to the wide space between those two types, which leave space for small size applications (van Eyk et al., 2017).

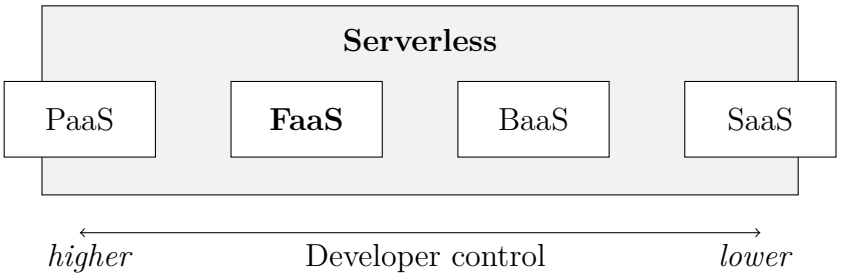

Figure 2.3.: The serverless model in the context of service models (taken from van Eyk et al. (2017))

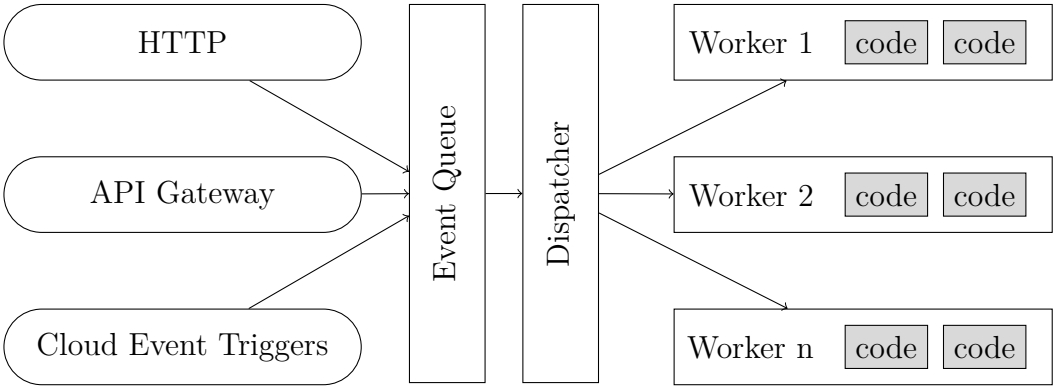

Figure 2.4.: Serverless platform architecture (based on Baldini et al. (2017))

Roberts and Chapin (2017) identify the handling of scaling as the key difference between PaaS and FaaS, as in PaaS scaling is still the responsibility of the consumer.

Figure 2.3 introduces Backend-as-a-Service (BaaS). BaaS is a vendor-specific solution to ease e.g. user management or data storage by replacing and out-sourcing server-side components with third-party services (Roberts and Chapin, 2017). BaaS is mentioned for completeness, but not further considered as it is rather an external API possibly implemented by using the serverless model.

In principle, every FaaS implementation is based on the architecture displayed in Figure 2.4. Multiple event sources exist to trigger a cloud function. The event is stored in a queue until a worker starts the execution of the cloud function code and finishes. The dispatcher ensures that a worker is available and is able to execute the latest code. The system itself must manage, start and stop the cloud functions, while taking care of resource allocation. (Baldini et al., 2017)

Implementations

Several in maturity and feature-support varying implementations of FaaS exist. Section 2.3 will discuss further the implementation of FaaS by the major cloud service providers separately. In addition to these mostly proprietary solutions, some open source FaaS implementations are discussed here.

The Apache OpenWhisk project provides the execution environment for the professional FaaS implementation of IBM Cloud Functions (more in Section 2.3.3). The architecture of Apache OpenWhisk requires all functions to run inside of Docker containers. This enables the execution of any code of any programming language as long as it can be containerized with Docker.

Next to Apache OpenWhisk, the kubeless[1] project uses a kubernetes cluster to execute cloud functions. Moreover, the Docker-LambCI[2] projects goal is to provide the exact same environment as AWS Lambda even using the official deployment tools.

Additionally, the OpenLambda project is a reference architecture for serverless platforms and poses software engineering challenges in the design space to find fast, flexible and efficient solutions. (Baldini et al., 2017, Hendrickson et al., 2016)

Spillner (2017a) provides an overview of the current FaaS runtimes with a focus on the Python programming language. Additionally, the author implements a new modular solution in the attempt to study and improve FaaS implementations.

2.2.4 Use-Cases

The serverless architecture is especially suited for resource intensive operations (Baldini et al., 2017). The classical example is manipulation of media files, although it is not the only one. Example use cases include:

Image Manipulation

Images allow for multiple image operations on the same image. Depending on the application, various image operations like brightness adjustments, rotation and/or contrast stretching is done.

Scaling / Compression / Conversion In the web, the time which a customer spends on an e-commerce website correlates with the page load speed, which includes media files like images (Nah, 2004). As compressed images only show limited detail, scaling of images is desirable to provide a thumbnail quickly and then show the details later in a high-resolution shot.

Watermarking Another typical use-case is the watermarking of an image. This can be a (transparent) logo that get applied on top of the image or as a text link to a website to indicate the author of the image.

[1] `https://github.com/kubeless/kubeless`
[2] `https://github.com/lambci/docker-lambda`

Also, the watermarking can be embedded dynamically during file access to include meta-data or invisible tracking information. The tracking information could be used to research the distribution of such a file. An example is the use of Digital Rights Management (DRM) to identify the original purchaser. Also, the use of stenography falls in this use-case.

Bots

The event-driven model suits the use-case of bots well as they only have to respond to incoming requests (events). As an example, Yan et al. (2016) builds a serverless chat bot using Apache OpenWhisk.

Internet-of-Things

Additionally, the field of Internet of Things (IoT) benefits from serverless cloud functions as work can be offloaded on-demand. The expected low load does not require a provisioned VM-based solution as its utilization will be low most the time.

High Performance Computing

High performance computing typically involves a task that can be parallelized to lower the time till the result is available - although the same amount of computational resources is required in total. This fits the FaaS model, as the parallel execution is done automatically through scaling.

The work of Jonas et al. (2017) use AWS Lambda to implement and measure multiple different high performance computing scenarios. While they identify the storage throughput as a major bottleneck, their implementation is only 17% slower than a spark cluster, excluding the initial start and setup time of the spark cluster.

Spillner et al. (2017) gives the example of password cracking to utilize the parallelization in FaaS. Similar, also detection and recognition of objects or faces in images and videos can be done quickly using FaaS.

2.3 Cloud Service Providers

For gaining an overview of the state-of-the-art of the field of serverless computing, first an overview of the FaaS offerings is needed.

On the market there are many companies that offer FaaS in the public cloud. However, for this work only the four biggest companies are selected. According to RightScale (2018), these are AWS Lambda, Google Cloud Functions, IBM Cloud Functions and Microsoft Azure Functions. The term cloud service provider is used as an umbrella them.

The selected cloud service providers all offer public clouds. According to Mell and Grance (2011), in a public cloud, "the cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider".

Besides offering a similar service, their cost model as well as their availability differs. Next to the description on the websites of the cloud service providers, Lloyd et al. (2018), Malawski et al. (2017b) provide an overview.

The selected cloud service providers are ordered in alphabetical order, without any intention of ranking. The stated information is up-to-date at the point of writing, but subject to change as the cloud service providers may revise their offerings in the future.

2.3.1 AWS Lambda

Amazon was the first company to bring serverless computing to a greater audience in November 2014 under the name of AWS Lambda. (Amazon Web Services, Inc., 2018a)

Out of all chosen providers, AWS Lambda providers the greatest amount of flexibility. The memory usage can be specified in steps of 64 MB starting from 128 MB up until 3008 MB (a little less than 3 GB). Moreover, many more configuration options like concurrency and chaining of cloud functions is possible.

While there is no information on the amount of CPU cycles allocated per memory configuration, it is stated that CPU power is allocated "proportional" to the chosen amount of memory.

The benefit in AWS Lambda lies also in the integration with other AWS products, which can also be considered as vendor lock-in if being used.

2.3.2 Google Cloud Functions

Since its launch in February 2016, Google Cloud Functions was in beta status till mid-June of 2018. Now, the service is ready for production use. (Google LLC, 2018a)

The options for configurations are limited as well as sparse in possible choices compared to the other selected cloud service provider, although enough options exist to run and monitor an application.

In opposite to the providers, Google is the only cloud service provider to give more detailed information about the amount of allocated CPU cycles as shown in Table 2.1.

Memory	128 MB	256 MB	512MB	1024 MB	2048 MB
CPU cycles	200 MHz	400 MHz	800 MHz	1,4 GHz	2,4 GHz

Table 2.1.: Allocated CPU cycles per memory configuration in the Google Cloud Functions environment according to Google LLC (2018b)

2.3.3 IBM Cloud Functions

IBM Cloud Functions uses the open-source Apache OpenWhisk project (see Section 2.2.3) internally for their FaaS implementation. That has the benefit, that a local[3] Apache OpenWhisk implementation can be operated as well. Therefore, IBM Cloud Functions is the only implementation without a direct vendor lock-in.

Apparently, the memory is limited to 512 MB, although according to the configuration more is possible. However, by using the memory management of Docker, more memory is addressable for cloud function without instant function termination. On the flip side, the performance is degraded as will be shown in Section 3.4.1.

2.3.4 Microsoft Azure Functions

Microsoft Azure Functions is the FaaS offering from Microsoft embedded in the Azure Cloud.

In opposite to the other cloud service providers, Microsoft Azure Functions does not require the user to specify the amount of required memory during the deployment of functions - making it the only true serverless implementation.

Instead only the used peak memory consumption per invocation is measured and billed for the complete execution period. Additionally, the BTU of Microsoft Azure Functions is the lowest compared to the other cloud service providers with 1 ms.

2.3.5 Cost Model

The cost model in serverless computing can be difficult to decipher and hard to model (Eivy, 2017).

The consumer is billed based on two components: the amount of function invocations and the resource consumption. The function invocations can be easily counted, while the resource consumption is more complex to understand.

[3]For the local environment, the Apache OpenWhisk project in revision *e7d2c7cc131ce84c90c75cbeccc2f6243676f543* from 1st of February 2018 is used as hosted on `https://github.com/apache/incubator-openwhisk/`.

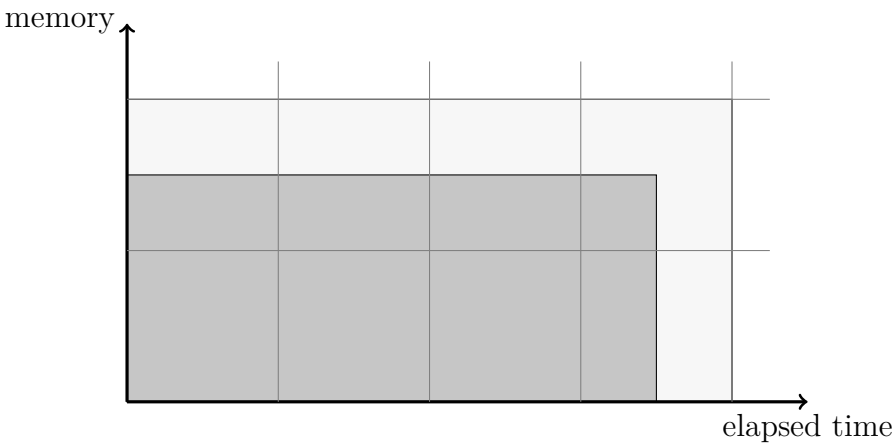

Figure 2.5.: Cost model of serverless computing in the two dimensions of memory and time visualizing the difference between billed and actual resource consumption

The resource consumption also consists of the two components: the cloud function execution time and the memory allocation. The allocatable memory is defined during the deployment of the cloud function and serves as a maximum value. The associated unit is GB-seconds, that is the amount of memory provisioned multiplied by the cloud function execution time.

For both components, not the actual usage is being billed, but an approximation based on a step-function. While the memory is specified during deployment, the cloud function duration rounded up to the nearest billable time unit (BTU). The BTU is the smallest billable time interval (100 ms for most service providers, see Table 2.2). Thus, billing is done in blocks.

Figure 2.5 visualizes the cost model using blocks. The small boxes represent on the x-axis one BTU and a memory step on the y-axis. The dark gray rectangle exemplifies the actual usage consumption of a cloud function, the light gray rectangle the billed consumption.

It becomes clear that a higher resource consumption is billed than actual used. Therefore, a too deep cloud function decomposition and low cloud function execution times can become expensive quickly. Kuhlenkamp and Klems (2017) introduce a “waste” metrics to measure the unused potential.

Cost Example

A function is running with 512 MB for 220 ms. It is called 10 times per second. The BTU is 100ms, therefore rounding up function execution time to 300 ms:

$$resource\_units = \frac{10 invocations}{s} * \frac{3.600s}{hour} * \frac{512\ MB}{1024 \frac{MB}{GB}} * \frac{300ms}{1.000 \frac{ms}{s}} = 5.400 \frac{GB-seconds}{hour}$$

So, the function costs 5.400 GB-seconds per hour in operation. Depending on the GB-second price, the total cost can be calculated (USD $\approx 0,09$). Additionally, the price for the 36.000 function invocations come on top.

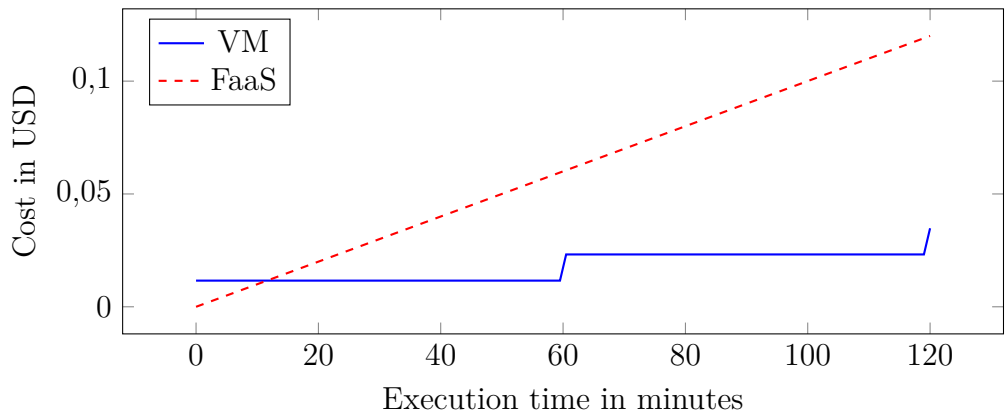

Figure 2.6.: Comparison of the cost for running a serverless function continuously and the AWS EC2 t2.micro instance - both having 1 GB of memory available. (Lambda=0,40; EC2=0,348 USD per hour)

Outside of these factors, the cloud service providers might bill additional service like storage for the cloud functions, incoming and outgoing traffic, advanced log processing, etc. At the moment, no cloud service provider charges for code deployments, provisioning and possible cold-starts of cloud functions.

All cloud service providers offer a free tier, which allows to run the first GB-seconds on their infrastructure for free; e.g. 400.000 GB-seconds per month on AWS Lambda. This seems a lot, but as Eivy (2017) points out that amount is fastly consumed. 400.000 GB-seconds are 4,6 days of possible resource consumption with 1 GB of memory - the actual resource consumption is lower due to previously explained block cost model. Also, the free tier of 1 million free requests on AWS Lambda is essentially only one request every 3 seconds, far away from a reasonable public cloud service. (Eivy, 2017)

However, the serverless cost model is attractive for low-usage applications. "In particular, the code may be scaled to zero where no servers are actually running when the user's function code is not used, and there is no cost to the user. This is in contrast to VM-based solutions where the user is charged even during idle periods." (Baldini et al., 2017)

Finally, a comparison to the cost model of VM-based solutions needs to be drawn. Under the assumption that on AWS the EC2 t2.micro instance is comparable to the AWS Lambda with both providing a maximum of 1 GB of memory and 1 vCPU, the VM-based solution is cheaper in the long-run assuming full utilization and therefore continuous execution of AWS Lambda as Figure 2.6 shows.

Notice that this is an extreme example hugely in favor for the VM-based solution due to the continuous, constant load. As this load is unrealistic in the practice, serverless on-demand computing becomes of interest.

	AWS Lambda	**Google Cloud Functions**	**IBM Cloud Functions / Apache OpenWhisk**	**Microsoft Azure Functions**
Memory Min	128 MB	128 MB	128 MB	128 MB
Memory Max	3008 MB	2048 MB	512 MB	1536 MB
Timeout Max	5 min	9 min	5 min	10 min
Billing Interval	100 ms	100 ms	100 ms	1 ms
GB-second price in USD	0,00001667	0,00001650	0,00001700	0,00001600
Memory Allocation	Fixed	Fixed	Fixed	Dynamic
Natively Supported Languages	C# Go Java Node.js Python	Node.js	Java Node.js PHP Python Swift ...	C# F# Node.js
HTTP Invocation	✓	✓	✓	✓
HTTP plus Authentication	✓	—	✓	✓
Free Tier (One time / Periodical)	✓/ ✓	✓/ ✓	✓/ ✓	✓/ ✓

Table 2.2.: Comparison of the major cloud service providers based on the data available at the point of writing

2.3.6 Summary

The offerings by the cloud service providers is summarized in Table 2.2. In general, the offerings of the cloud service providers are similar but differ in the details with only Microsoft Azure Functions providing a very different product. The only common runtime in between them is the Node.js JavaScript runtime at the time of writing.

In case that a cloud function executes longer than the timeout or tries to allocate more memory that is available, the cloud function gets terminated. The cause for the termination is accessible via the log files of the respective FaaS implementations.

Also, a basic free tier is offered by all cloud service providers to port and test application without charge. Spillner (2017c) takes the free tier offering of not only serverless computing, but also storage to a new level. The current product offerings are not economically feasible in his opinion, so that many hobby and smaller projects can run essentially for free in the cloud. (Spillner, 2017c)

Chapter 3

Benchmark

According to van Eyk et al. (2017), one of the challenges of serverless computing is the "focus on Cost/Performance". The authors add: "A deployed function can be migrated from one cloud to another seamlessly, if the new cloud offers a better fit between requirements and achieved performance", due to being designed platform-agnostic.

At the point of writing, multiple public cloud service providers have launched their own FaaS implementation of the serverless cloud computing execution model.

In order to gain knowledge of the state-of-the-art FaaS offerings in the public cloud, a benchmark is needed. A benchmark is one way of measuring the performance of a system (Li et al., 2013). The benchmark presented in this chapter is used to evaluate these offerings. A special focus is put on the relationship between performance and cost.

Most of the in the following discussed findings have been published in Back and Andrikopoulos (2018). While recasting, also new insights are gained and presented. All data related to the benchmark is on Github[1].

3.1 Related Work

The research in FaaS benchmarking is still limited considering how recent serverless computing still is. Though, benchmarks for VM-based solutions along with possible cost reductions exist, for example by Gómez Sáez et al. (2015).

The authors of Villamizar et al. (2015) execute a performance benchmark of three full cloud scenarios including a gateway, database and load-balancer comparing a cloud application implementations as monolith, microservices and serverless. They find that infrastructure cost can be reduced significantly using the serverless computing model. However, their benchmark is limited to one application as well as one public cloud service provider.

[1] `https://github.com/timonback/faas-mubenchmark`

Also, Spillner et al. (2017) observe the performance of cloud function. However, the focus lies in gaining an understanding of the cost model, not in executing a benchmark to compare different cloud service providers.

Additionally, the performance between FaaS and PaaS has been analyzed by Albuquerque Jr et al. (2017). Although the authors only use one FaaS implementation (AWS Lambda).

The authors of Costradamus Kuhlenkamp and Klems (2017) list as one of their findings that "cost calculations should not exclusively be based on analytical models but include real measurements". They focus on tracing the cost of a serverless based software through AWS Lambda on a per-request level. Using it, the computational "waste" related to the specific cost model in serverless computing is shown. Although no generic functions, but rather a specific use-case is used, unexpected performance findings are revealed.

In both works, Lloyd et al. (2018) and Lee et al. (2018) execute explicit benchmarks on public FaaS implementations. Both provide meaningful insights in terms of performance, where they use more fine-grained tasks and focus on the technical aspects like concurrency, latency and throughput.

Especially with the work of Malawski et al. (2017a), the connection to high performance computing becomes clear - for which serverless computing is suited as well. The findings are a great addition to this work, although are missing the relationship between performance and cost.

3.2 Design

The goal of the benchmark is next to comparing the performance of the cloud service providers verifying previously made hypotheses. These are presented first, before selecting different algorithms. These are then implemented as a cloud function for the benchmark.

3.2.1 Hypotheses

Besides checking the performance of the cloud service providers FaaS implementation's, prior benchmark hypotheses (BH) have to be verified. These are listed below.

BH-1: The observed performance is invariant to the cloud function implementation

It is expected that executing a cloud function with twice as many instructions also takes twice the amount of time to complete. The underlying hypothesis is that each cloud function gets a fixed share of the resources. The performance should neither vary during the execution nor between multiple runs.

BH-2: If memory is coupled to the performance, then more allocated memory leads to a faster execution

Both, AWS Lambda and Google Cloud Functions document that the amount of allocated memory is proportionally linked to available CPU cycles. Thus, it is expected that for these cloud service providers, more memory leads to a faster execution of the same function.

BH-3: The ranking of performance between cloud service providers is identical for all cloud functions

When comparing a cloud function with the same configuration between different cloud service providers, then the ranking of the performance is expected to be the identical. The underlying architecture and hardware of the cloud service providers is not known and the ranking is not known. However, a change in the ranking is not expected.

BH-4: The billed duration matches the execution time

When executing a cloud function, it is expected that only the actual execution time is billed. External overhead caused by the cloud service provider for preparing and invoking the function is not expected to be billed or kept to a negligible minimum.

For clarification: The cloud service providers bill the execution time of a cloud in BTUs as explained in Section 2.3.5. The to the next BTU rounded *billed duration* relates to the measured execution time that is used to calculate the billed duration. If a function runs for 450 ms as measured by the function itself internally, it is not expected that it gets billed for 600 ms.

BH-5: For short-living cloud functions, the billed duration is constant independent of the parameter values to the functions

Starting a cloud function and loading the dependencies takes time. For short-living cloud functions, preparing takes longer than executing it.

3.2.2 Functions

As the underlying architecture and hardware in serverless computing is unknown, functions with different characteristics are used for the evaluation. The benchmark includes four functions, which are described in more detailed in the next paragraphs. Additionally, a sleep function is used as a baseline.

The chosen functions vary in the amount of required resources in terms of computations and memory as shown in the overview in Table 3.1.

The benchmark is implemented in the JavaScript run-time environment Node.js, since it is - the only one so far - supported by all cloud service providers natively. All functions are deterministic dependent on the input. No randomness is involved. The output is not further used than validating the result during testing.

Function	Computation	Memory
Fast Fourier Transform	High	High
Matrix Multiplication	High	Medium
Pi	High	Low
Union-Find	Low	High
Sleep	Low	Low

Table 3.1.: Relative resource requirements of the benchmarked cloud functions

All files required to execute the benchmark are included in the public Github repository at `https://github.com/timonback/faas-mubenchmark`.

Fast Fourier Transform

The Fast Fourier Transform (FFT) is listed in the Top 10 Algorithms of 20th Century by the IEEE journal Computing in Science & Engineering and has a major importance in signal processing. With it, a signal is sampled over a time or space period into the frequency domain and back again by the Inverse Fast Fourier Transform. Some computations are significantly less intensive in the frequency domain than in the original domain, which makes the frequency domain attractive for image analysis and manipulation.

The used Cooley-Tukey method uses a divide and conquer approach. For the execution of the benchmark, the *fft-js* library[2] in version 0.0.11 is used.

The input parameter specifies the number of discrete signal samples, which already for small number input creates a computational-heavy load.

Matrix Multiplication

Another common operation in Computer Science is the multiplication of matrices (MM). The used algorithm is a naive, unoptimized implementation in the complexity of $O(n^3)$.

The input parameter specifies the length of each of the two square matrices. So, if the parameter doubles, the size of one matrix quadruples as well does the complexity of the matrix multiplication.

Pi

An example of a scientific computation poses the exact calculation of pi (PI). One way of obtaining pi is by using the Leibniz formula. Due to the continuous amount of additions, subtractions and divisions, this algorithm is heavy on computational resources while using almost no memory.

Leibniz formula: $1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}$

The input parameter specifies the total amount of additions and subtractions.

[2] `https://www.npmjs.com/package/fft-js`

Union-Find

Union-Find algorithms (UF) are used to partition data into discrete sets. One use is component labeling in images. It is a simple algorithm, but dependent on the data input, has high requirements on memory and many unpredictable memory accesses, which lead to many cache-misses.

For the benchmark, the *union-find*[3] package in version 1.0.2 is used.

The input parameter specifies the number of vertexes used. They get linked in sets of 10.

Sleep

The sleep function (S) is only used as a reference to measure the difference between the expected execution time and the overhead by the cloud service provider that is being billed. The wrapper code around the sleep function is expected to take a couple milliseconds as well, but is expected to be neglectable. The sleep command is expected to be independent of CPU cycles as well.

The input parameter specifies the amount to sleep in milliseconds.

3.3 Setup & Execution

As previously listed, the major cloud service providers are being benchmarked. The timeout of each cloud function is increased to 5 minutes (Google Cloud Functions timeout is set to 9 minutes). Each cloud function is tested in 5 memory configurations[4]: 128 MB, 256 MB, 512 MB, 1024 MB, 2048 MB. These values are chosen, as they are supported by all cloud service providers and their value spacing.

The deployment of the cloud functions is provider dependent. To ease the deployment process, reduce complexity and have a better overview of the configurations, the *Serverless Framework*[5] is utilized. With the bindings to the big cloud service providers, the deployment is simplified to a single configuration file and one command per cloud service provider: `serverless deploy` or `sls deploy` in short. It is assumed that previously an account has been created at the cloud service provider and the authentication and configuration details are available for the Serverless Framework as specified in their documentation.

Since the cloud service providers expect different bindings for the cloud functions, for each provider a custom minimal wrapper was created, which reads the passed-in parameters, calls the appropriate function and returns the result. The called algorithm is identical for every cloud service provider. In appendix A.2, more information and a code excerpt of the wrapper is provided.

[3] `https://www.npmjs.com/package/union-find`

[4] As stated in Section 2.3.3, Apache OpenWhisk only supports up to 512 MB. Moreover, Microsoft Azure Functions does not use a pre-configured memory limit, but bills according to the maximum usage.

[5] `https://serverless.com/`

Function	Parameters
Fast Fourier Transform	8.192 16.384 32.768 65.536 131.072 262.144 524.288 1.048.576 2.097.152 4.194.304 8.388.608 16.777.216
Matrix Multiplication	100 200 300 400 500 600 700 800 900 1.000 1.500
Pi	64 128 256 512 1.024 2.048 4.096 8.192 16.384 32.768 65.536 131.072 262.144 524.288 1.048.576
Union-Find	65.536 131.072 262.144 524.288 1.048.576 2.097.152 4.194.304 8.388.608 16.777.216 33.554.432
Sleep	0 2 4 8 16 32 64 128 256 512 1.024 2.048 4.096 8.192

Table 3.2.: Used parameters for the benchmarked cloud functions

The benchmark was executed across 3 consecutive working days in the end of April 2018[6]; resulting in three measurements per cloud function and parameter for each cloud service provider. The used script for testing the functions at the different cloud service providers is provided in appendix A.1. Each cloud function is previously warmed up by querying so that a cold start as explained by Baldini et al. (2017), Roberts and Chapin (2017) is avoided.

The cloud functions are invoked from a local machine at the University of Groningen using the `curl` command, although the location of the function invoker does not affect the measurements. The cloud function deployment location is kept comparable (AWS Lambda: *us-east-1*, Google Cloud Functions: *us-central-1*, Microsoft Azure Functions: *Central US*) with the exception of IBM Cloud Functions (*United Kingdom*) due to limitations of the free tier.

The used parameters for the functions are listed in Table 3.2, which are whole numbers, mostly of the log2 subset. The log2 subset is chosen to reduce the number of steps needed to reach a maximum value, while keeping an equal spacing between the parameters. The range of parameters are chosen by author in a way to max out the available resources.

The baseline test using the open source Apache OpenWhisk project was run on a local machine with an Intel i7-6700HQ (4x2.6GHz) with 8GB of memory. The installation itself run in VirtualBox machine running Ubuntu Linux 14.04 LTS with 4 GB of the total 8 GB allocated.

3.4 Results

The results to the benchmark are presented in the following paragraphs. Out of all available graphs, only the relevant ones are picked and presented along verifying the Hypotheses.

[6]Google Cloud Functions was still in beta status at that time

All collected measurements are drawn as a scatter plot. Additionally, a mean function for each provider shows the average values. Only data points with a successful execution of the cloud function are included. If a request fails due to a timeout or insufficient amount of memory, no data point is drawn, although the duration till failure may have been measured.

As a shorthand for referencing the functions, the FunctionMemory notion is introduced, where Function $\in$ {FFT, MM, PI, S, UF} and Memory $\in$ {128, 256, 512, 1024, 2048}. For example, FFT512 stands for the Fast Fourier Transform function with 512 MB of memory.

For clarification, the measured duration is the one reported in the log files of the respective cloud service provider and also used for billing purposes. Latency of the network is not measured, thus the location of the cloud function as well as the invoker does not matter in this benchmark.

3.4.1 Hypotheses

In the following paragraphs, the hypotheses are verified.

BH-1: The observed performance is invariant to the cloud function implementation

The complexity related to BH-1 is expressed by the input parameter of the cloud function. Figure 3.1 shows that for each cloud service provider, the performance is invariant for PI2048, since the duration follows a linear trend. The same is valid for the other cloud functions. Therefore, the hypothesis holds.

BH-2: If memory is coupled to the performance, then more allocated memory leads to a faster execution

Out of the selected cloud service providers, AWS Lambda and Google Cloud Functions do couple the performance to the amount of allocated memory according to their documentation. (Amazon Web Services, Inc., 2018c, Google LLC, 2018b)

The relationship between memory and performance is shown in Figure 3.2. First of all, the performance is invariant with respect to complexity as already confirmed by BH-1. The differences in successful executions related to the memory configuration is discussed in Section 3.4.3. However, changes in the performance ranking between the memory configurations exists.

The performance based on the memory configuration is shown more clearly in Figure 3.3. The cumulative execution time of the FFT function across memory configuration is plotted with $k \in [8.192; 131.072]$ to ensure the same number of data points for each configuration and cloud service provider.

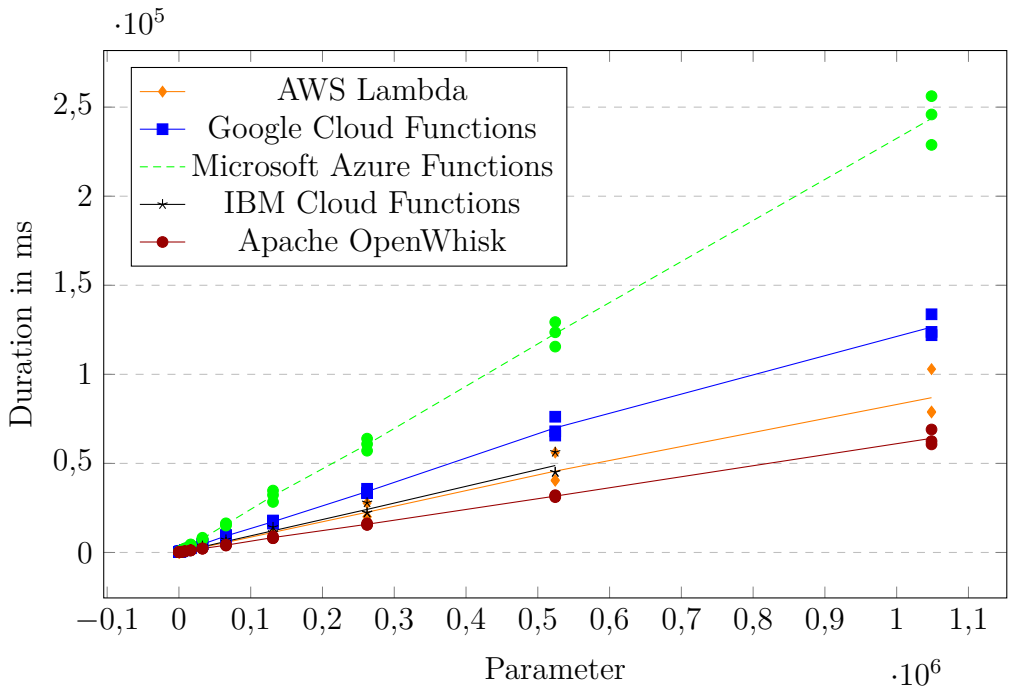

Figure 3.1.: Measured successful durations of PI2048 across all cloud service providers

AWS Lambda and Google Cloud Functions clearly result in faster execution times with more available memory - as it is coupled with performance. The data points of Google Cloud Functions flatten out, as the increase of performance is not linear as shown in Table 2.1. The exact speed-ups are shown in appendix A.3.

For the other cloud service providers, the variation in cumulative execution times is negligible. They provide a constant, fair share of the available resources and only the consumed memory is charged, while the number of CPU cycles is constant.

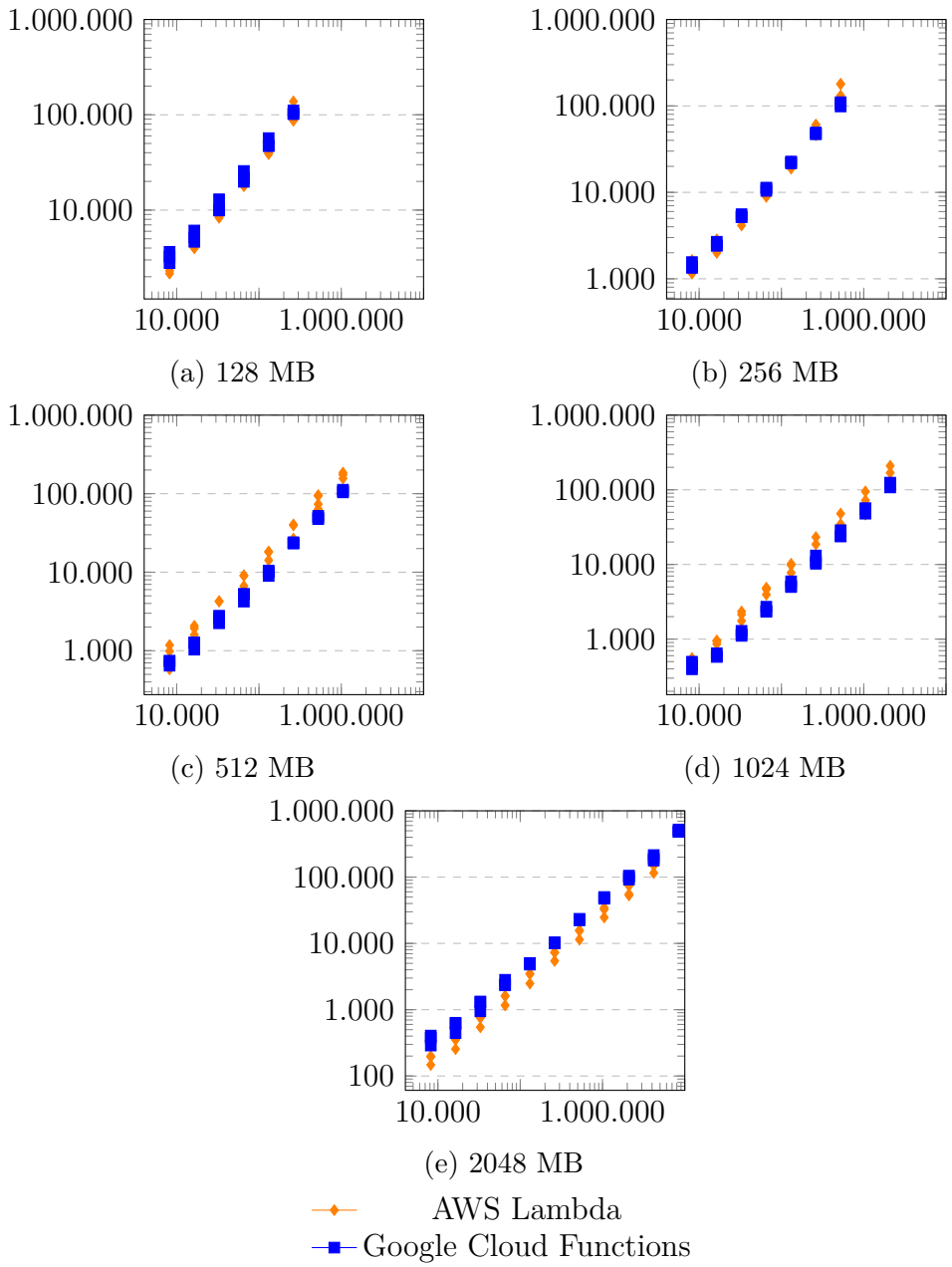

Figure 3.2.: Measured successful durations of FFT in different memory configuration

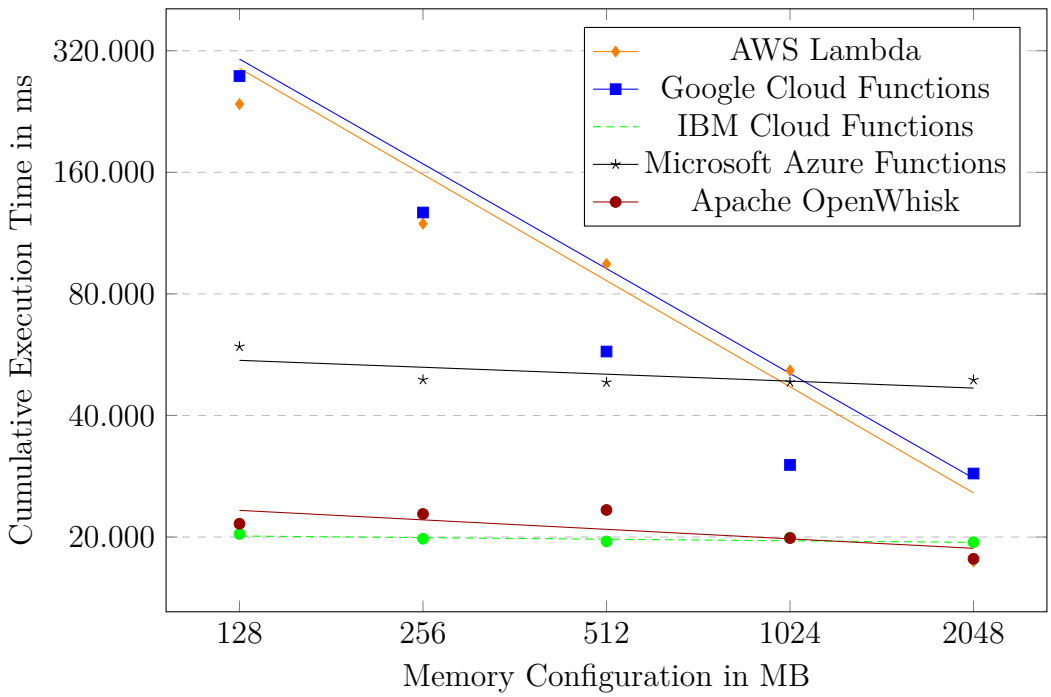

Figure 3.3.: Cumulative execution times of FFT in different memory configurations using only the successful executions ($k \in [8.192; 131.072]$)

BH-3: The ranking of performance between cloud service providers is identical for all cloud functions

Figure 3.1 gives a first impression of the ranking of the cloud service providers. To verify the hypothesis, also the ranking between the other cloud functions need to be considered. For a fair comparison, the maximum memory configuration of 2048 MB is used so that the CPU is not limited on any platform (see BH-2).

The ranking is shown in Figure 3.4, where a lower cumulative execution time indicates a cloud service provider with higher performance. Of the public clouds in the 2048 MB memory configuration, IBM Cloud Functions has the best performance, closely followed by AWS Lambda. A significant gap towards Google Cloud Functions and Microsoft Azure Functions exists. Also, the local Apache OpenWhisk performs well, as long as not too much memory is required (UF2048).

BH-4: The billed duration matches the execution time

To obtain qualitative numbers about the duration of a cloud function, the Sleep function (S) is used. Since no calculations are involved, external influence is minimized. The measured durations are shown in Figure 3.5.

First, AWS Lambda seems to offer the FaaS cloud with the least amount of billing overhead. The Sleep function, which is constant in time, finishes in all memory configuration the fastest on AWS Lambda. In Figure 3.5 it seems that the other cloud service providers get closer to the actual expected execution time for higher parameters and a longer execution, but this is only attributed to the log scale of the graph.

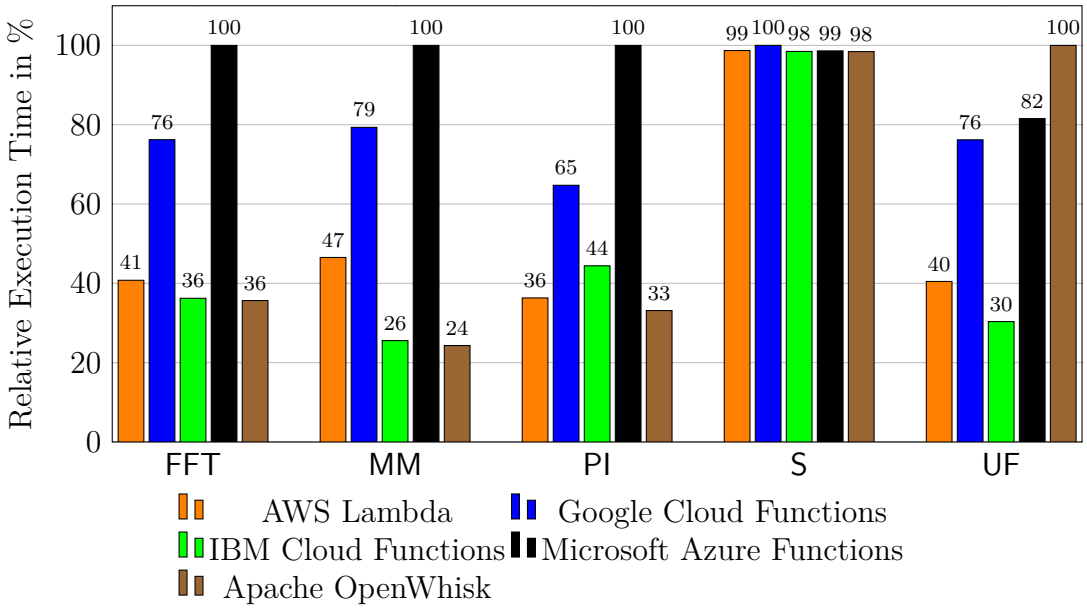

Figure 3.4.: Ranking of the cloud service providers of the cloud functions FFT2048, MM2048, PI2048, S2048 and UF2048. Lower relative execution time is better.

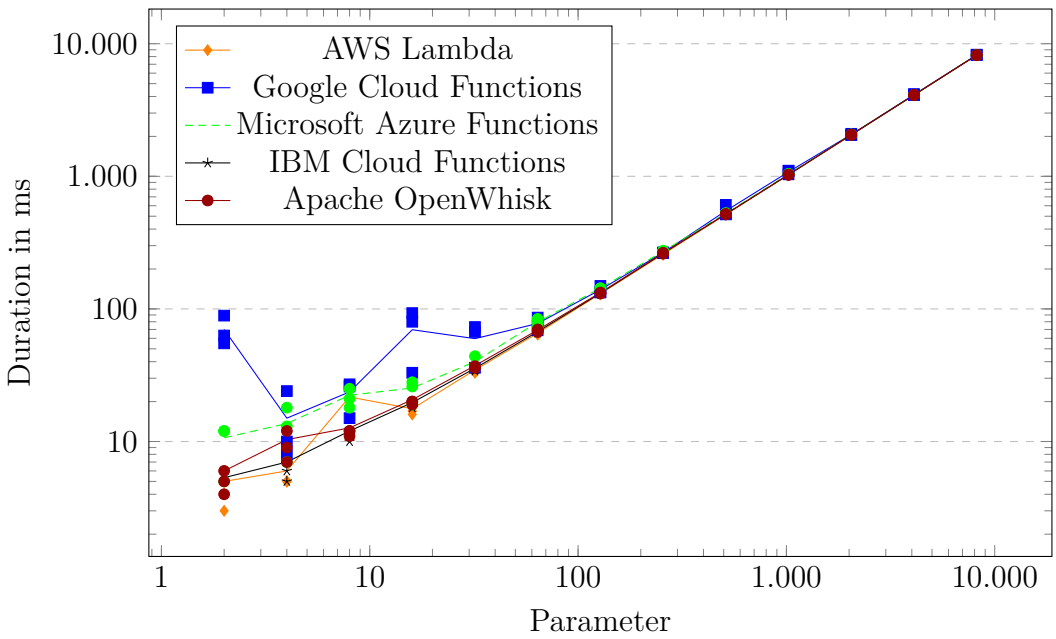

Figure 3.5.: Measured successful durations of S2048 across all cloud service providers

Configuration	AWS Lambda	Google Cloud Functions	IBM Cloud Functions	Microsoft Azure Functions	Apache OpenWhisk
S128	265,82	2.597,61	1,63	22,40	6,18
S256	62,46	1.589,33	12,40	57,72	24,10
S512	41,96	726,93	1,79	20,04	12,06
S1024	31,62	757,52	2,03	14,63	15,96
S2048	12,31	851,3	2,40	18,75	5,72
$mean(MSE)$	81,03	1.304,54	4,05	26,71	12,80

Table 3.3.: Mean Square Error (MSE) for linear regression to the observed data of S per memory configuration per cloud service provider (taken from Back and Andrikopoulos (2018))

Second, related to BH-3, the amount of allocated memory does not seem to have an effect on the delay. Comparing Figure 3.5 (S2048) with Figure A.1 (S128) reveals that they are identical. Since S is not executing anything, this is expected.

Third, for an unknown to us reason, Google Cloud Functions require at least 10 ms to finish. Also, Google Cloud Functions takes in average[7] 51ms longer to execute S, which is half of a BTU.

The variation of the measured durations in Table 3.3 is expressed using the Mean Square Error explained e.g. by Montgomery and Runger (2010).

So, the billed duration matches the execution time in most cases. Only Google Cloud Functions is not as mature and bills significant overhead time, where the other cloud service providers show that it is not necessary.

BH-5: For short-living cloud functions, the billed duration is constant independent of the parameter values to the functions

This hypothesis holds for most functions (excluding S). It mainly depends on the input to the function, so that the calculation does not get too complex. This is best seen with PI with small parameters. The graphs in Figure 3.6 follow the expected hockey-stick function.

This effect is especially visible for short-living cloud functions with only a couple BTUs execution time, but always present.

Therefore, the hypothesis holds. Although it should be noted, that in these cases not the full capacity potential is used. Kuhlenkamp and Klems (2017) refers to this as computational “waste”.

[7] Averaged difference of the means of AWS Lambda and Google Cloud Functions

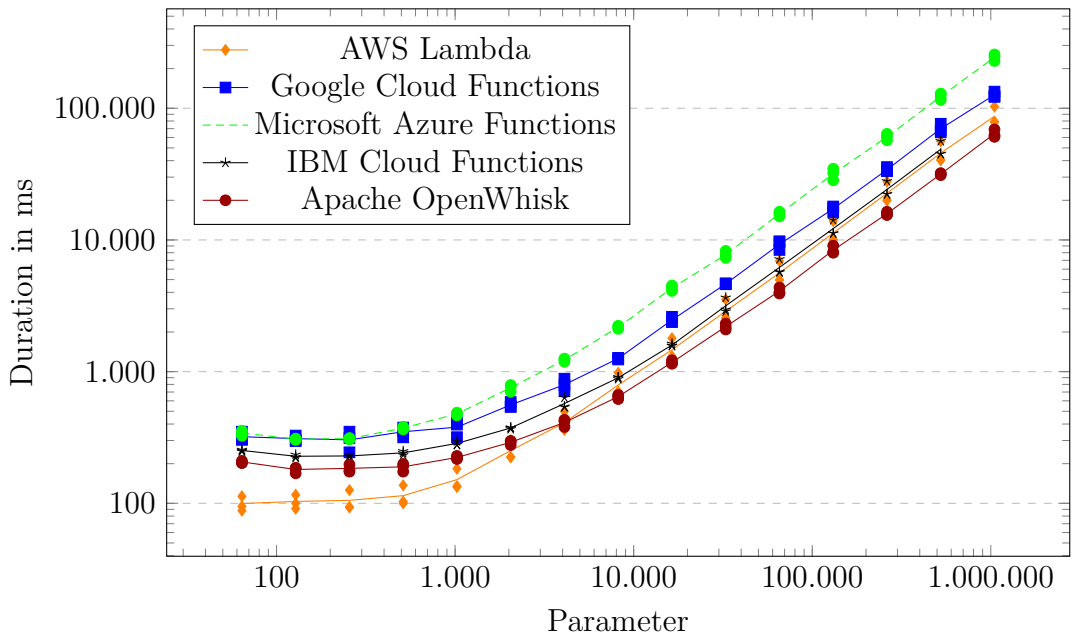

Figure 3.6.: Measured successful durations of PI2048 across all cloud service providers

3.4.2 Memory Consumption

Only AWS Lambda provides data on the peak memory consumption during the execution of a cloud function. The other cloud service providers only indicate an over-allocation of memory by terminating the cloud function.

The memory limitation of Apache OpenWhisk is mentioned in Section 2.3.3. Due to the memory handling, the execution time explodes using more complex functions or parameters, although the cloud function still executes instead of terminating. This effect is visible in Figure 3.7 for $k >= 4.194.304$.

The consumed memory per cloud function and parameter is shown in appendix A.5 in Table A.3, Table A.4, Table A.5 and Table A.6. In general, dependent on the function, the range of the memory consumption depends on the dependencies and the memory requirements of the algorithm.

3.4.3 Cloud Function Termination

Observations in Figure 3.2 of BH-2 give already a hint concerning the unexpected termination of cloud functions. Cloud functions do get terminated if they allocate more memory than specified during the deployment.

While in Figure 3.2a only the lowest 6 parameters result in successful invocations, Figure 3.2d with a higher amount of allocated memory shows 10 successful invocations for the same function.

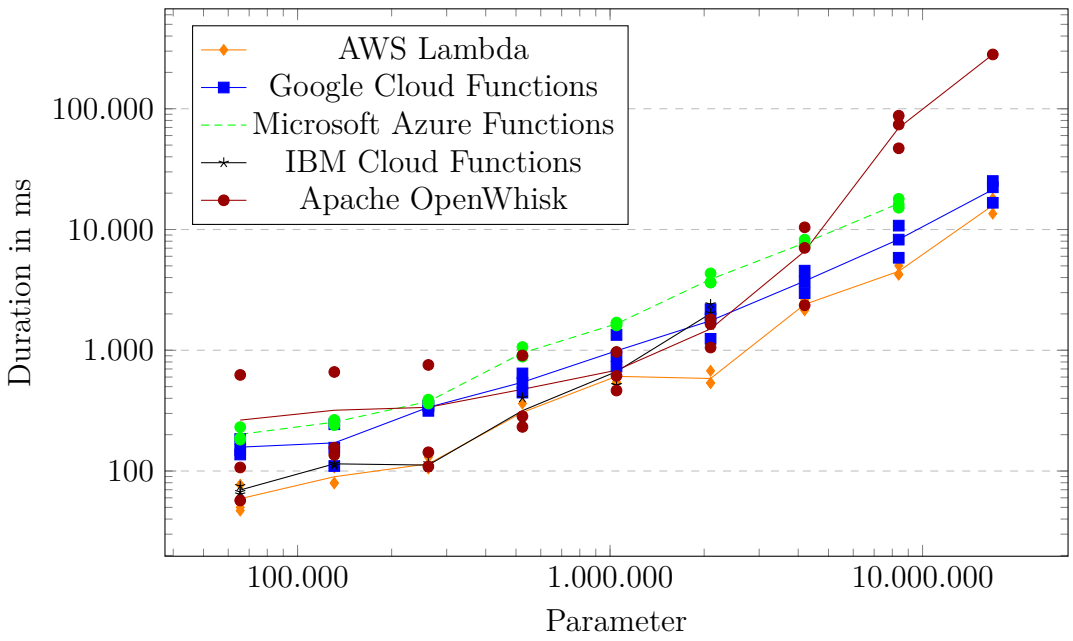

Figure 3.7.: Measured successful durations of UF2048 across all cloud service providers

$k =$	**AWS Lambda**	**Google Cloud Functions**	**IBM Cloud Functions**	**Microsoft Azure Functions**	**Apache OpenWhisk**
[8.192; 131.072]	15	15	15	15	15
262.144	15	15	12	15	15
524.288	12	12	9	15	15
1.048.576	9	9	0	15	15
2.097.152	6	6	0	15	13
Total:	∼ 86,7%	∼ 86,7%	∼ 71,1%	100%	∼ 98,5%

Table 3.4.: Successful executions of FFT across all memory configurations per parameter k (taken from Back and Andrikopoulos (2018))

Table 3.4 puts it the observation into numbers for FFT. The higher the input parameter k, the higher is the memory consumption as shown in Section 3.4.2 and appendix A.5. Thus, cloud functions requiring more memory than available get terminated.

Microsoft Azure Functions do not have this problem, as there the memory gets provisioned dynamically. Moreover, due to previously mentioned memory management of the local Apache OpenWhisk implementation, the cloud function does not get terminated, but slows down in performance (see Section 2.3.3).

	AWS Lambda	Google Cloud Functions	IBM Cloud Functions	Microsoft Azure Functions	Apache OpenWhisk[9]
FFT128	236,900s	277,600s	21,100s	59,247s	22,400s
GB-s ($BTUs*\frac{1}{8}$)	29,613	34,700	2,638	7,406[8]	2,800
FFT256	119,900s	128,100s	20,500s	49,029s	23,600s
GB-s ($BTUs*\frac{1}{4}$)	29,975	32,025	5,125	6,129[8]	5,900
FFT512	95,500s	58,300s	20,300s	48,309s	24,000s
GB-s ($BTUs*\frac{1}{2}$)	47,750	29,150	10,150	6,039[8]	12,000
FFT1024	52,500s	30,800s	20,700s	48,338s	20,600s
GB-s ($BTUs*1$)	52,500	30,800	20,700	6,042[8]	20,600
FFT2048	17,900s	29,500s	20,200s	48,982s	18,300s
GB-s ($BTUs*2$)	35,800	59,000	40,400	6,123[8]	36,600
Total GB-s	195,638	185,675	79,013	31,739	77,900
GB-s Price	0,00001667	0,00001650	0,00001700	0,00001600	0,00001700
Total Cost	0,00326129	0,00306364	0,00134322	0,00050784	0,00132430
Relative Cost	100%	94%	41%	15%	41%

Table 3.5.: Measured cumulative duration in seconds of FFT over different memory configurations and cloud service providers with $k \in [8.192; 131.072]$. Respecting the pricing related to the GB-seconds unit, a total cost in USD is calculated.

3.4.4 Cost

The incurred cost of executing the benchmark is shown in Table 3.5. It only considers the resource consumption and excluded the price for the number of cloud function invocations. The additional price for every started 1 million requests after beyond the Free Tier is $0,20 at AWS Lambda and Microsoft Azure Functions, $0,40 at Google Cloud Functions and $0,00 at IBM Cloud Functions.

Table 3.5 shows for each memory configuration of FFT the execution duration in seconds followed by the GB-s unit taking into account the amount of available memory per function with input parameter $k \in [8.192; 131.072]$. This choice k allows for the same number of measurements for each configuration, although the execution duration is relatively short. Comparing the cell numbers in BTU rows against each other is possible as the same amount of computation was done - even if the memory was not used to the full extent. Finally, the total GB-s and the cost are calculated along with a comparison of relative cost to each other.

[8] Microsoft Azure Functions determines the memory automatically, which for Fast Fourier Transform with k parameter is always below 128 MB, therefore multiplying with $\frac{1}{8}$.

[9] Apache OpenWhisk is deployed in a local machine. Since IBM Cloud Functions uses Apache OpenWhisk, the same pricing information is used.

First, Microsoft Azure Functions with the lowest total cost gains attention. However it should be noted, that Microsoft Azure Functions determines the cost for the used memory dynamically, resulting in an unfair comparison to the other cloud service providers for the total price.

Second, IBM Cloud Functions and Apache OpenWhisk also offer low prices. That is due to the fact that the performance is not coupled to the amount of available memory as observed in BH-2.

Third, AWS Lambda and Google Cloud Functions lie closely together in terms of total cost. Still, the performance per memory configuration varies between as well as within the cloud service providers.

Notice, that for e.g. on AWS Lambda FFT2048 executes faster and is cheaper than FFT512 and FFT1024. Also, FFT128 and FFT256 have the same price, while FFT256 finishes twice as quick. Therefore if BH-2 applies, a higher amount of allocated memory can be faster and cheaper.

Provider ranking

The cloud service providers can be ranked in three different categories, which is done in the next paragraphs.

When Microsoft Azure Functions is adjusted for the dynamic memory consumption by multiplying each execution time with the according memory allocation as done for the other cloud service providers, the total GB-s is 190,119, which results in an adjusted total cost of $0,00304191 (relative cost: 93%).

Duration For the duration, per cloud service provider the minimum cell value for the cumulative execution duration is compared. That is FFT2048 on AWS Lambda with 17,9s. Therefore, AWS Lambda provides the platform with the fastest possible execution environment.

GB-s For the GB-s unit, per cloud service provider the minimum cell value for GB-s is compared. IBM Cloud Functions provides the highest performing execution environment while charging the lowest amount of GB-s in the FFT128 configuration.

Total Cost The GB-s are directly translated into total cost, as the GB-s prices are closely by each other (variance: $4{,}1e^{-14}$). Thus, IBM Cloud Functions is the expected to be the cheapest.

3.5 Limitations

The executed benchmark has limitations that need to be mentioned.

First, the amount of collected data points is limited. Although three consecutive measurements per function, memory configuration and parameter on each cloud service provider were taken, more data points would be beneficial to confirm the findings with a higher certainty. Still, they fit the pictures compared with previously, unpublished measurements.

Second, only Node.js was used as it is the only programming language supported by all cloud service providers. As more languages become widely available, another benchmarks specific for the programming language needs to be executed.

Third, the free tier on all platforms was used. It is not expected that this has an impact on the measurements or the performance, but neither is this disproved.

Fourth, the serverless framework is used for deployment. It is a convenient tool taking care of the deployment steps. The exact deployment configuration was not researched, but it is not expected that the framework has an effect on the measurements or performance.

Fifth, Google Cloud Functions was still in beta status during the execution of the benchmark. As it is marked as production ready now, the underlying architecture might have changed requiring a new run of the benchmark.

3.6 Lessons Learned

By means of the benchmark, the validity of the made Hypotheses was checked. Additionally, more observations were made, which are summarized in the following list.

1. The performance per cloud service providers follows a linear trend. For some, also the performance is coupled to the amount of provisioned memory.

2. The relationship between cost and performance differs between the cloud service providers - for some even significantly.

3. The surrounding eco-system is an important factor in deciding for a cloud service provider. Even for the deployment of cloud functions, multiple other factors and products are relevant like log processing and data storage. However, this can introduce a vendor lock-in.

4. Only Microsoft Azure Functions provides a real serverless FaaS implementation. The other cloud service providers require the consumer to specify the maximum amount of allocatable memory during deployment.

5. The maturity especially of Google Cloud Functions requires a beta status due to measured mean square error and overhead billing of half a BTU.

6. More benchmarking is required as the measured performance also depends on the executed cloud function. The rankings of the cloud service providers switch between memory configurations as well as between computational and memory heavy algorithms. Also, other factors like I/O as in Lee et al. (2018), Lloyd et al. (2018) play a role.

7. Cloud function do terminate due to insufficient memory instantaneously. A previous warning or chance to save made progress is not possible nor intended.

It seems that AWS Lambda provides the most stable environment with also the richest eco-system around it. Moreover, Google Cloud Functions is taking steps in the area of serverless computing, but the platform is correctly labelled with a beta status. IBM Cloud Functions with the usage of Apache OpenWhisk makes the use of a hybrid private and public cloud an interesting scenario as Apache OpenWhisk can be hosted in a local environment, while offering the lowest price. However, the maximum amount of memory is a limiting factor resulting in high error rates. Finally, Microsoft Azure Functions is providing the only real serverless FaaS implementation, although the author found the handling of logs and information about the billing troublesome.

3.7 Conclusion

The developed and executed benchmark reveals that the compared serverless computing FaaS implementation do differ not only on the specification, but also in performance and induced cost.

To decide on the best cloud service provider, the requirements of the specific cloud application needs to be known. As the cloud service providers improve their products, a migration at a later stage might be interesting. Although serverless computing ideally allowing for easy migration between clouds, a vendor lock-in exists, thus making a later migration difficult.

Also, this benchmark confirms the finding of that a higher initial price linked with more resources can pay off through shorter execution times as mentioned in Kuhlenkamp and Klems (2017). But, in can only be stressed that “calculations should not exclusively be based on analytical models but include real measurements”.

Chapter 4

Decision Model

As shown in the previous chapter, the serverless architecture provides a reliable platform to execute functions in the cloud. Scaling is handled effortlessly, transparent and on-demand.

In the long-run and assuming a constant load, VM-based solutions are cheaper due to better utilization compared to FaaS approaches as Figure 2.6 shows. However, perfect utilization is rarely the case in reality. Therefore, a decision model is required to make an educated decision about which cloud computing execution model or combination of execution models is most cost-efficient for a given cloud application and load.

Previous works covered in Section 4.1 have presented decision models, but not to the extent of a hybrid decision model. The focus lies on creating a simple decision model with the relevant decision criteria and straightforward in usage. The decision model is derived by including the relevant parameters of the cloud applications load, FaaS and VMs-based deployment models as well as lessons learned through the Benchmark like BH-2.

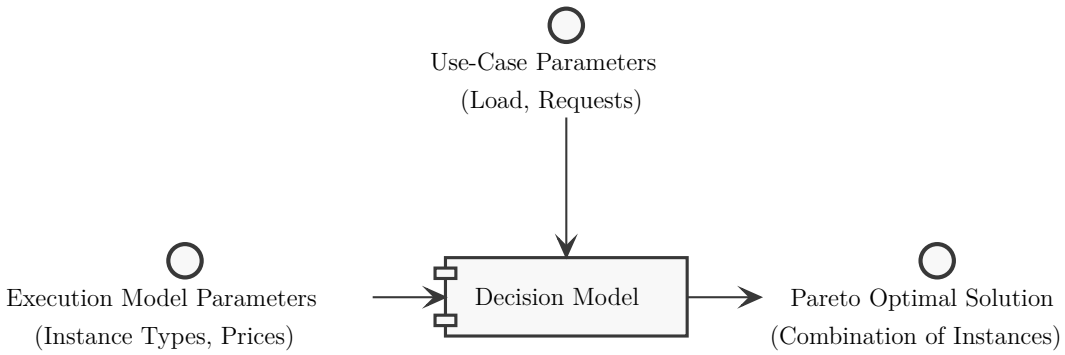

Figure 4.1.: Overview of the decision model

This chapter explain the assumptions, parameters and strategy for the hybrid deployment model to minimize cost, before ending with a short summary. While the model itself is generic, most parameters are custom to a specific use-case. The next chapter shows the implementation of this decision model in form of a simulator.

4.1 Related work

The work by Albuquerque Jr et al. (2017) provides a first step into the decision model, by showing the differences between FaaS and PaaS. However, they run a comparison based on scenarios without providing assistance in the decision process. Moreover, their scenarios do not contain a hybrid solution.

CostHat by Leitner et al. (2016) provides an advanced deployment cost model for cloud applications. The model incorporates multiple cloud functions (API calls), IO and more. The outcome of the work is an Eclipse IDE extension to infer costs caused by changes in the source code. This enables a What-If analysis while identifying the parameters for performance and optimal distribution of load between serverless cloud functions and VM-based solutions is not tackled. Still, the work is valuable, as the impact of changes can be tracked in a graph structure throughout a project.

The experiment by Jonas et al. (2017) indicates that the FaaS implementation by AWS Lambda can hold up with the computational requirements of high performance computing. The authors notice that the serverless implementation is around two times more expensive to run. It can be interesting to reorganize the project towards a hybrid deployment solution to be more cost-effective.

In Sharma et al. (2011), the authors propose *Cost-aware* and *Cost-oblivious* provisioning model to reduce the cost of a cloud application in the AWS cloud using the EC2 product. They implement a greedy algorithm to achieve significant cost reductions.

4.2 Assumptions

Every model is a simplification to capture reality in a reasonable scope. As a consequence, not all possible cases are handled by a model, but only a realistic subset.

In order to reduce complexity, the following decision model assumptions (DMAS) are being made. These assumptions are needed for the model to be an accurate representation of the world, while still be realistic.

DMAS-1: The load is known and countable

For the decision model to be able to find the most cost-effective deployment configuration, a typical load for an application must be known. The load contains all incoming requests over time that need to be handled. It can be artificially created by a mathematical function or be a previously recorded load.

Further, the requests have to be countable, as this puts a limit on the maximum number of requests. This is required to find quickly a first - probably not ideal - solution.

DMAS-2: Resource consumption per request is fixed

Similar to the known load distribution, also the resource consumption per request needs to be known. By combining all requests, the complete resource requirements over time have to be known.

The resource consumption per request can differ, due to different input parameters, external dependencies, etc. in the real world. However, an upper limit is always known and can be used as a pessimistic value for this purpose.

DMAS-3: All requests can be handled interchangeably by FaaS and VM-based solutions

As mentioned before in Table 2.2, the cloud service providers set upper limits on the execution of requests on FaaS. These limitations constrain the type of incoming requests more than the VM-based offerings.

The types of limits might change in the future, however at the current moment the main limiting factor is the maximum execution time per request. Other limitations include the maximum memory available.

DMAS-4: VM provisioning time is zero

For the decision model, VMs are usable starting from the moment required. This allows a simplification that VMs have a constant execution duration. It avoids the handling of BTUs for VMs. While in reality, one VM would continue in operation after each BTU, in this model the VM gets terminated and a new one provisioned in the next step.

DMAS-5: Load-balancing and -redirection is perfect

Finally, it is also assumed that the load-balancing and -redirection of incoming requests is perfect. Incoming requests will only be redirected to instances that are able to handle the request exactly at that moment. No request is queued. The exact strategy is laid out in Section 4.4.

Assumption Justification & Limitations

These assumptions are valid to make, as they are limiting the reality in a minimal way, while keeping the complexity of the decision model low.

DMAS-1 and DMAS-2 ensure that sufficient data is known in the model, so that a decision can be reached in reasonable time. Dynamic changes during usage of the model are forbidden and require a recalculation of the model.

DMAS-3 is required so that a fair comparison between FaaS and VM-based solutions is possible and that all requests are handleable by both service models. In a more complex version of the model it should be considered to lift this constraint to handle any kind of request. However, it shall be noted that such a request would require more than 3008 MB of memory or take longer than 5 minutes to complete. However, serverless computing is usually not suited for those tasks anyway.

In DMAS-4 it is already argued that zero provisioning time is similar to just keep a VM-based instance running. This assumption could be lifted, however would make the decision model unnecessary aggravated.

Finally, DMAS-5 allows to model the load-balancer in an idealistic way. Still, if the incoming requests are similar enough in resource consumption, then a perfect balancer already exists using the round-robin technique to distribute the load evenly. However, the load balancer needs to know the load of the provisioned instances.

4.3 Parameters

The parameters of the decision model can be split into two categories. Some are specific to the cloud computing execution model, others to the individual cloud application.

The decision model does not directly take into account the specifics of a cloud function. Rather, inside each request the resource requirements are specified, which abstracts away any specific function or input parameters.

Also, the granularity of a cloud function does not need to be defined. While from the programming perspective it can be a single algorithm or data modification, it can also be a complete program with enormous amount of code instructions. For the decision model, it is indifferent.

4.3.1 Load

Any cloud application reacts to incoming requests. The load are the combined incoming requests over time. The term originates out of the area of networking, where the load is defined on a more fundamental level of network traffic consisting of packets.

In this model, a request is described as a tuple being *Request(incoming_time, duration, memory)*. Assuming that $\mathcal{R}$ contains all possible *Requests*, then $\mathcal{L} \subseteq \mathcal{R}$ represents any imaginary load $\mathcal{L}$.

If a load is known from the application, then it can be used directly. Otherwise, as will be shown in Section 5.2.2, artificially generated load from periodic functions is also well suited.

4.3.2 Instance Types & Pricing

With price being the main objective to minimize, the decision model needs information about the different cloud computing execution models, configuration and their price. This data includes the offerings of the cloud service provider of choice with the computational and memory resources it provides as well as cost.

Part of the pricing is also the BTU. Besides the assigned price per BTU, FaaS and VMs differentiate in their flexibility. While the BTU is fixed for both, the BTU of FaaS is considerably smaller. Moreover, FaaS bills in interval steps up to the maximum execution time as only used execution time is billed as explained in Section 2.3.5.

An instance type is represented as *InstanceType(parallel_requests, memory, btu, btu_max, price)* with $\mathcal{T}$ containing all available instance types. As an instance type is only a template, multiple instances of the same instance type can be created with different instantiation times. An instance is defined as *Instance(type, starting_time)*.

Every instance type provides a specific amount of computational resource (*parallel_requests* that can be handled and the maximum amount of *memory* available), has a *price* per *btu* and has a maximum execution time *btu_max*. *parallel_requests* is the number of requests, that the instance type can handle in concurrently. The *memory* element is the total amount of available memory, which is shared between all requests that are being processed concurrently. Obviously, a FaaS instance type can only handle one request in total, which means one request concurrently. Therefore, the instance cost in FaaS is $price * \frac{request\_duration}{btu}$.

For VMs, the *btu* and *btu_max* values are identical in this model[1]. A VM is terminated after reaching the execution time of *btu_max*, which is acceptable due to DMAS-4. The instance cost of any VM is equal to the *price* of the instance type.

4.4 Strategy

The common understanding of load-balancing is that the load is equally fair distributed between the available resources. However, "in cloud computing a critical goal is minimizing the cost of providing the service. This leads to a

[1] Vice-versa if *btu* and *btu_max* are never identical for FaaS, the instance type category (FaaS or VM) can be determined without the need for an additional element

different meaning of the term load balancing; instead of having the load evenly distributed among all servers, we want to concentrate it and use the smallest number of servers". (Marinescu, 2017)

The goal of the decision model is to find the combination of instances in $\mathcal{I}$ - based on the instance types in $\mathcal{T}$ - which handle the load $\mathcal{L}$ (consisting out of the requests out of $\mathcal{R}$) with the lowest total cost $\mathcal{C}$. The total cost $\mathcal{C}$ is the sum of the individual costs of each instance out of $\mathcal{I}$.

It can thus be seen as a variation of the vehicle routing problem in logistics which is a variation of the traveling salesman problem (Cormen et al. (2009)): Which trucks (instances) should be used to process all stations (requests) the most cost-effective?

This is a combinatorial problem, as only after trying all combinations, the truly optimal solution is found. In complexity theory, these problems fall into NP-hard, while deciding whether a better solution exists is NP-complete.

As finding the truly optimal solution is computation expensive and the problem size explodes, the strategy is to use a greedy algorithm fitted to this problem and exploiting DMAS-3.

4.4.1 Steps

Following are the steps of decision model shown.

Get or Generate a Load

First, a load $\mathcal{L}$ is required that needs to be handled. This can be a recorded load or also an artificial load pattern as later presented in Section 5.2.2.

Finding the First Solution

For each request, try to assign it to the first VM-based instance in $\mathcal{I}$, that is able to handle the request. To max-out the utilization, iterate though the available instances always from the beginning. If no VM-based instance is available, create a new VM-based instance using an instance type template in $\mathcal{T}$ and add it to the end of the ordered set $\mathcal{I}$. Any kind of balancing of requests between the available instances is ignored.

The simplified code in Algorithm 4.1 demonstrates the basic working of handling incoming requests with VM-based instances. Further details including autoscaling are hidden.

The first solution is an all VMs-based solution. Due to DMAS-3, it is ensured that all requests can be handled.

Algorithm 4.1 Finding an initial solution and cost with only VM-based instances

```
Require: requests is sorted by incoming time
  vm_instances ← {}
  for r ← 0; r < requests.length; r ← r + 1 do
      request ← requests[r]
      handled ← 0
      for i ← 0; i < vm_instances.length; i ← i + 1 do
          vm_instance ← vm_instances[i]
          if vm_instance.handle(request) ≠ 0 then
              handled ← 1
              break
          end if
      end for
      if handled = 0 then
          vm_instance = InstanceVm(...)
          vm_instance.handle(request)    ▷ DMAS-3: all requests are handleable
          vm_instances ← vm_instances + vm_instance
      end if
  end for
  return vm_instances, run_cost_calculation(vm_instances)
```

Finding More Solutions

The previously constructed set $\mathcal{I}$, contains all instances needed to handle the load. Now, all VMs-based instances are iteratively replaced by FaaSs-based instances and requests are reassigned to the new instance. This is shown in the simplified code in Algorithm 4.2. Performance differences between VM-based and FaaS-based instances and BH-2 potentially need to be considered as well.

The order of replacement is decided by the utilization of the VMs-based instances. VMs-based instances with a low utilization have are poor cost-benefit ratio and are thus replaced first.

Included in `run_cost_calculation()` is code to calculate the cost of the instances. Let n be the number of instances created in the first step, then there will be $n+1$ total solution configuration in $\mathcal{S}$. The i index represents the number of provisioned VMs ($i \in 0...n$). $\mathcal{S}_0$ is an all FaaS solution, $\mathcal{S}_n$ is an all VM-based solution and in between are hybrid solutions.

Pareto Optimality

A solution is Pareto optimal if any reallocation of resources worsens the utility in at least one criterion, while it may improve utility in a different criterion (Pardalos et al., 2008). Additionally, the Pareto front describes the set of solutions from a solution space that are Pareto optimal.

Out of all Pareto optimal solutions in $\mathcal{S}$, the solutions $\mathcal{S}_i$ are in the Pareto front $\mathcal{P}$, where the total cost of that solution $\mathcal{C}_i$ is the lowest: $\mathcal{P} = \{\mathcal{S}_i \in \mathcal{S} : \mathcal{C}_i = min(\mathcal{C})\}$.

Algorithm 4.2 Replace iteratively VM-based instances with FaaS instances

```
Require: vm_instances is sorted ascending in utilization
  instances ← {}, f_instances ← {}, costs ← {}
  for i ← 0; i < vm_instances.length; i ← i + 1 do
      vm_instance ← vm_instances[i]
      requests ← vm_instance.requests
      for r ← 0; r < requests.length; r ← r + 1 do
          request ← vm_instance.requests[r]
          f_instance ← InstanceFaas(request)
          f_instances ← f_instances + f_instance
      end for
      vm_instances[i] ← {}                                  ▷ Remove the instance
      instances ← instances + {vm_instances + f_instances}
      costs ← costs + run_cost_calculation(vm_instances + f_instances)
  end for
  return instances, costs
```

Pareto front graphs are shown in the next chapter, for example in Figure 5.7 and Figure 5.8. Compared to the commonly used Pareto front graphs, the y-axis is reversed so that the optimal configuration is where the y value is the lowest. Since the total cost is shown, this is arguably more natural to read.

4.4.2 Limitations

Due to the fact that a greedy algorithm is being used, depending on the input it can get stuck in a local optimum.

One of these situations is when always before a long duration request is coming in, a short duration request is blocking the instances. Then a new instance is required for the long duration request, which are better suited to maxing out the utilization of instances. However, the problem is less dramatic, as the instances with the lowest utilization are being replaced first.

The main limitation based on the assumptions is the perfect load- and request-balancer. However, for similar enough loads, a round-robin approach is justifiable. One mandatory requirement for the load-balancer is to be able send load transparently to either VM-based and FaaS instances.

4.5 Summary

The proposed decision model takes into account multiple aspects of a cloud application and the cloud computing execution to find a Pareto optimal deployment strategy. While the decision model is generic and can be used for any application, the concrete solution is tied to the chosen parameters.

The next chapter shows a the simulator, which implements and applies the decision model discussed in the previous sections.

Chapter 5

Simulator

The previous chapter creates the theoretical foundation for the simulator presented in this chapter. A simulator is one way to implement the Decision Model. As it has been shown by Calheiros et al. (2011), a simulator is a practical, useful tool to improve resource consumption.

In a broader sense, this simulator aims to provide tooling support for any cloud application and is available on Github[1]. Given the cloud application's specific parameters for the decision model, whose process of retrieving is explained in Section 5.4, a recommendation for the Pareto optimal deployment strategy is created.

This chapter is structured by first listing the requirements and design decisions made. Then the implementation of the simulator is shown as well as the method to obtain the cloud application's specific parameters for the decision model. Finally, the results are evaluated using different load patterns.

[1] `https://github.com/timonback/thesis-msc-simulator`

5.1 Requirements

Every software engineering project is developed under requirements that specify the mode of operation. Besides implementing the Decision Model, more requirements are added to create tooling support that is valuable and useful in the long-run.

The requirements are stated with the keywords *must*, *should* and *may* as proposed by Bradner (1997).

REQ-1: The simulator must finish within a reasonable time

One of the main requirements for the simulator is that it is fast enough in generating a new deployment strategy. Besides, any increase in performance lets the simulator run faster and allow for more simulations in the same time.

If the simulator is used reactively and iteratively to adjust the deployment strategy using the load of the last BTU for example, then the duration of one BTU is the limiting factor for what constitutes reasonable time.

REQ-2: The simulator must support at least the VM and FaaS execution model

The decision model itself does not separate between different execution models. Whether VM or FaaS is used is indifferent to it. However, the different nature of the execution needs to be considered. As this work focuses on comparing VM-based and FaaS instance types, at least these have to be considered.

REQ-3: The simulator must allow for performance adjustments between the execution models

The execution models of VM and FaaS vary. The performance and configuration options also have to be included in the simulator.

First, the simulator has to support that VMs can execute multiple requests at once, due to having multiple CPU cores and possibility even hyper-threading enabled.

Secondly, the performance between different instance types is different as Chapter 3 shows for the FaaS service model. The same holds also for VMs. Moreover, for some cloud service provider even different memory configuration have an influence (BH-2).

REQ-4: The simulator must support multiple, periodic load functions and their configuration

As it has been mentioned before in the Decision Model, the actual load of a cloud application is not always known. Possibility a pattern type is known. Therefore, the simulator must include multiple, different load patterns.

These load patterns should be configurable. Thus, it is also recommended to use period patterns as they are easy to understand and handle, while the output is predictable over multiple periods.

Besides the artificial load patterns, also custom load patterns must be supported. This can be split into two categories. First, the recorded load pattern that is typical to the cloud application of choice. The recorded load pattern then gets *replayed* in the simulator. Secondly, the simulator must be extensible so that new load patterns can easily be integrated as the simulator cannot contain all imaginable load patterns.

REQ-5: The simulator must support custom request configuration

While DMAS-2 still holds, each request needs to be configurable. Not every request is expected to be the same, although it can be. The duration of the requests as well as the required amount of memory may vary.

REQ-6: The simulator should support request randomization

Similar to REQ-5, a randomization might be desirable. The randomization exists in two dimensions: randomizing the number of requests per simulation step and randomizing the request duration.

Firstly, the artificial load patterns are so artificial that they follow mostly linear or curved functions. The idea of randomization is to add noise to let the artificial pattern appear more natural.

Secondly, the duration of a requests may vary. This can be due to the input parameter to the called cloud function. Also, external factors like the total utilization of the instance or other processes may have an unforeseen impact.

For both randomizations it is important to configure the randomization with parameters to fit the needs, while keeping the results reproducible.

REQ-7: The simulator should load pricing and instance type configuration options from an external source

The by the simulator supported execution models are not expected to change in the near-future. However, the offerings by the cloud service providers are likely to change as new offers come to the market and so will new instance types. With it, also the prices are subject to change due to newly introduced features and competition in the market.

Thus, this information should not be hard-coded into the simulator itself, but be loaded from an external source.

REQ-8: The simulator should persist or cache results

Although REQ-1 requires a reasonable simulator execution time, executing a simulation may take a while. Therefore, the results should be saved to readable caches. The cache can then be quickly loaded (useful for REQ-9 and REQ-10).

REQ-9: The simulator may plot the results for visualization purposes

Any simulation is data-heavy. The evaluation of the result is still done by humans and they process visual information better than lists of numbers. Also, visual representation allows to include more information in the same pixel area or as a proverb states: A picture is worth a thousand words.

As plotting the data is just a transformation of the result of the simulator, it is in an *optimal* requirement.

REQ-10: The simulator may provide an external web-interface module

Also, the web-interface is an *optimal* requirement. The goal is to provide a more interactive version of the simulator allowing for fast parameter adjustment cycles. A visualization component should then be integrated, similar to REQ-9.

5.2 Design Decisions

While designing and implementing the simulator, multiple decisions have been made. These are listed and explained in the next paragraphs.

5.2.1 Auto-scaling & Request Queuing

As known from practical experiments, auto-scaling in the cloud takes time. The decision model and thus also the simulator assume that provisioning of new requests is instant as stated in DMAS-4.

Requests Get Queued If Auto-scaling Is Disabled

Auto-scaling is a feature that allows adding or removing computational resources on the spot based on an automatic metric. In the situation that requests are coming in and no instance is available to handle it, the request is being queued. All requests in the queue are processed in FIFO order with higher priority over new incoming requests.

The queuing of requests only occurs when not enough instances are provisioned. Already with the first iteration of moving requests from the VM to FaaS instances, the queued requests get handled. All requests are handleable due to DMAS-3.

The default setting is to allow auto-scaling. Queuing of requests is only possible if auto-scaling is disabled and thus a potential violation of DMAS-5 for the all VM-based solution. Changing the parameter can result in the simulator not executing according to the decision model.

Minimum Number Of VM-based Instance Only During Incoming Load

As the provision time is zero (DMAS-4), forcing to have at least *min* instances running is only reasonable when also requests are coming in.

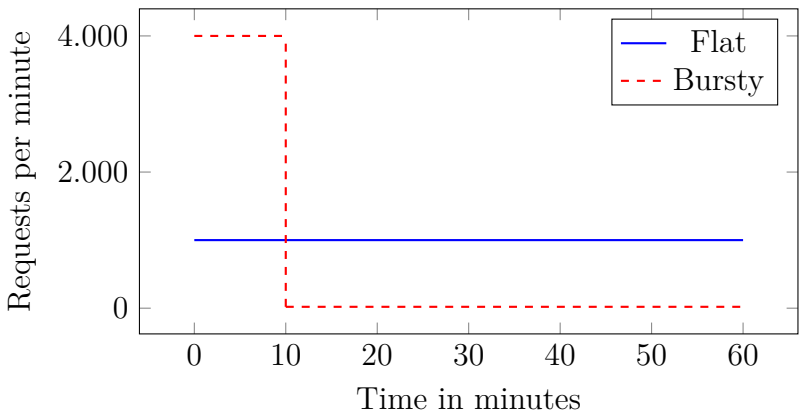

Figure 5.1.: Visualization of a constant and bursty load with the identical amount of total requests (based on Eivy (2017))

5.2.2 Load Patterns & Characteristics

Load is the distribution of incoming requests to a cloud application over time. The distribution of load can follow many different types of distributions. The simplest form is a flat load with a constant amount and evenly spread out requests for each time interval. Other distributions can follow wave forms or include breaks. In general, load is difficult to model using e.g. a Poisson distribution, as requests are expected to be uncorrelated and independent, which is not necessarily the case for bursty load.

Bursty load is not clearly defined in literature, but describes an increase in load over a short interval of time. An example is shown in Figure 5.1, where the total amount of requests is identical. One proposed detection metric is the measurement of the peak-to-average ratio (Khalil and Sun, 1992). Along with bursty load, also spikes may arise with one or more peaks.

The in the simulator included load patterns (REQ-4) are listed below. Every load pattern has an altitude and spacing parameter. The consumer is able to modify these parameters to their needs (REQ-4). In Figure 5.2, the chosen load patterns are shown. The number of generated requests varies between the load functions.

Figure 5.2 is not showing that a simulation is executed in a discrete space. There is not 0,3 of a request, either there is a full request or there is none. Still, the figure shows how the load patterns compare to each other.

The following paragraphs list the load patterns in more detail. These are chosen as they are common and periodic functions, which are used to model load.

Constant

The number of requests per simulation step is constant. This is the ideal load to handle with VM-based solutions.

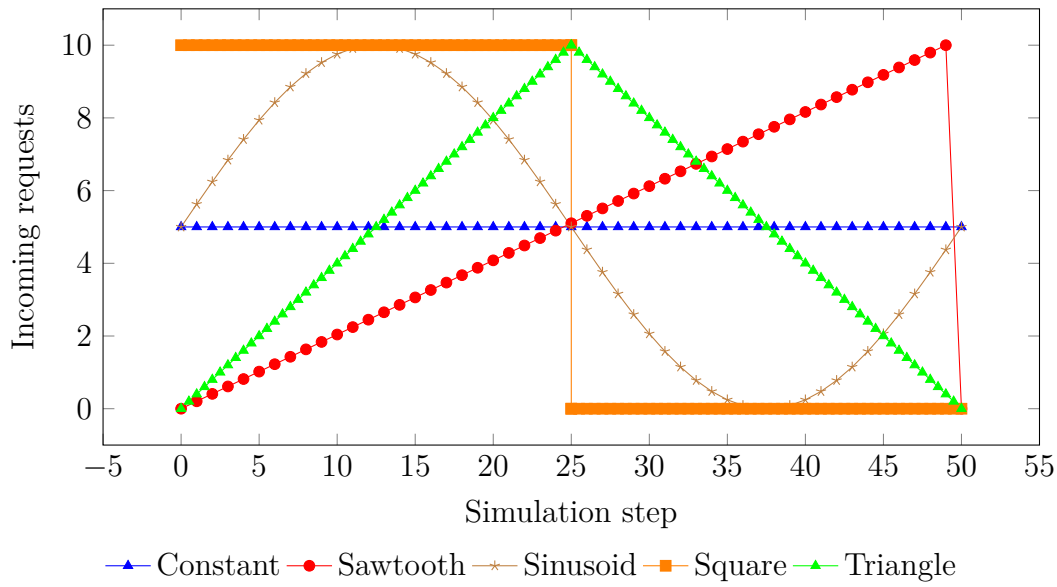

Figure 5.2.: Load distribution from different artificial load patterns

Sawtooth

The sawtooth load increases from zero linear up to *altitude*-many requests and then sharply drops down to zero. The sawtooth is suited for application, where the requests constantly increase until a certain point is reached.

Sinusoid

The sinusoid load pattern moves around the half of altitude in an increasing and decreasing wave pattern. It is a typical load pattern in many situations. A common situation is the day-night-cycle.

Square

The square load pattern is an extreme load pattern. For half of the time *altitude*-many requests are coming in, followed by no requests at all.

Triangle

The triangle load pattern is a mix out of the sawtooth and sinusoid load patterns. It monotonically increases until *altitude*-many requests and then monotonically decreases back to the starting value.

Normal Distribution

The normal distribution is special load pattern. This distribution is included, because using only the sigma parameter different levels of burstiness can be simulated.

In opposite to the other load patterns, in total four parameters are used: *sigma*, *spacing*, *number of samples* and *number of spikes*. The available *spacing* gets split up into *number of spikes*-many areas. In the center of each of them a normal distribution using the *sigma* parameter gets samples, so that in total *number of samples* are taken.

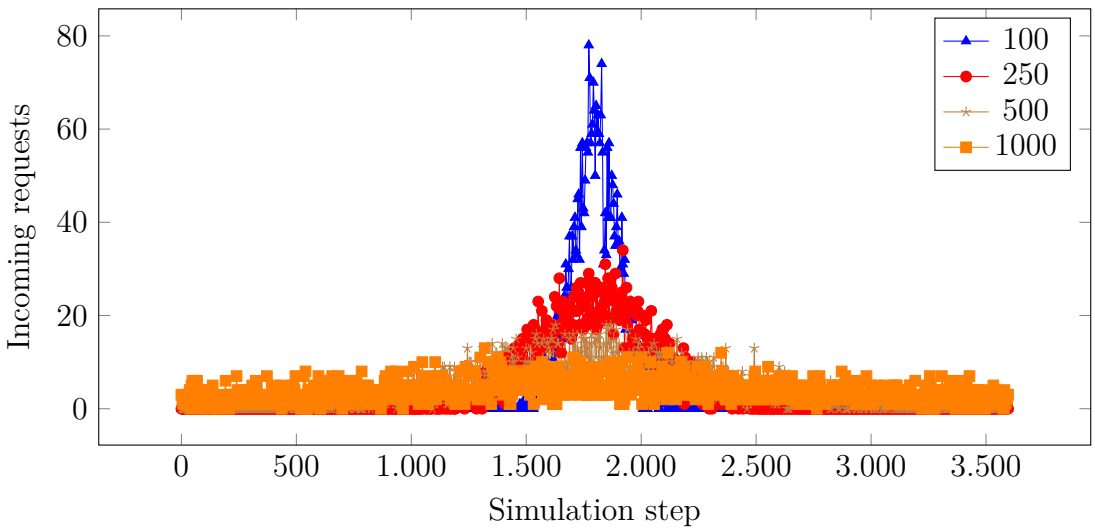

Figure 5.3.: Load distribution sampled (n=3.600) from the normal distribution using different sigma values

An example of this is shown in Figure 5.3, where multiple normal distributions with a spacing of 3.600 are sampled with 3.600 requests in total and one spike. The varied sigma parameter results in fundamental different peak shapes. Also note, that requests are sampled based on the normal distribution. The number of requests is therefore not necessarily continuous and contains jumps.

5.2.3 Technology Selection

For the implementation of the simulator, Python is the programming language of choice. The Python programming language has cross-platform support with reasonable performance. It is ideal for prototyping applications. Adding missing features is straight-forward (REQ-4).

Moreover, since Python is easy to learn, the chance of somebody wanting to use, extend and learn from the build simulator is higher. For this reason, the simulator is also documented. Anyone can have a look at the software and modify it. Also, unit testing and typing of the function parameters is including.

Finally, plotting the result (REQ-9) and offering an interactive web-interface (REQ-10) is simple without the need for many external dependencies that make the structure unclear.

5.3 Implementation

All of the Decision Model Assumption (DMAS) discussed in Section 4.2 are also valid for the simulator. Furthermore, some features have been added to not only implement the Decision Model, but also reflect the insights gained from the Benchmark.

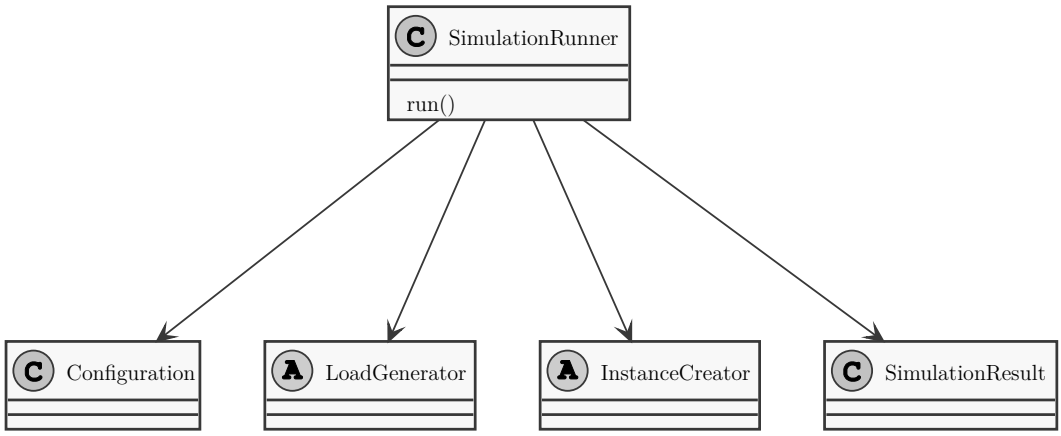

Figure 5.4.: Simplified architecture of the implemented simulator

The simulator is started via `python main.py` or the interactive web-interface via `python serve.py`, whose front-end is reached via `http://localhost:8888/index.html`. It is assumed that the dependencies are installed ideally in a virtual environment[2] via `python setup.py install`.

5.3.1 Architecture

The simplified class diagram of the simulator is shown in Figure 5.4. At the core of the simulator is the `SimulationRunner`, which uses the other components and executes a simulation via the `run()` method.

The input for the `SimulationRunner` is the `Configuration`, which contains all configuration to run the simulation, including parameters for the decision model and load that is generated via the `LoadGenerator`. The computing instances get created in the `InstanceCreator`, which both `InstanceCreatorFaaS` and `InstanceCreatorVm` implement (REQ-2). The final output is the result of the simulation: `SimulationResult`.

5.3.2 Decision Model

The decision model is implemented within the `SimulationRunner`. In the first step, the simulator creates only instances with the `InstanceCreatorVm` during the execution of `_simulate()`. Next, the FaaS instances of `InstanceCreatorFaaS` `_patch()` and replace the VM-based instances stepwise according to the decision model.

To take into account BH-2, a mapping is required to model the influence of the allocated memory to performance. This is done in form of a logarithmic function: $y = a + m * log(memory)$. With the amount of allocated memory as input, a performance factor is determined to adjust the cloud function's execution time. Obtaining the parameters is explained in Section 5.4.1. If BH-2 does not apply for the selected cloud service provider (e.g. IBM Cloud Functions or Microsoft Azure Functions), then a is set to one and m to zero.

[2] `https://virtualenv.pypa.io/en/stable/`

5.3.3 Features

The features of the simulator are covered in more detail in the next paragraphs.

Configuration

The configuration of the simulator is done via `simulator/configuration.py`. It is a Python class that can also be modified during run-time before a simulation is started. It contains multiple configuration blocks that are presented in the following paragraphs.

First, the duration (end - start) of the simulator can be configured as shown in Listing 5.1. It is advisable to change the duration in multiples of BTUs to ensure a fair comparison.

```
        # Timing limits of the simulation
        self.simulation_start = 0  # in seconds
        self.simulation_end = 3600  # in seconds
```

Listing 5.1: Configuration for the duration of the simulation. The unit is seconds.

Next, the VM-based and FaaS instance types can be configured as shown in Listing 5.2 (REQ-3). These parameters are explained in Section 5.4 as well as how to obtain them.

The performance factor is a log function, because of BH-2. If instead the performance is independent of the amount of memory, then set *faas_performance_a* to 1 and *faas_performance_m* to zero.

```
        # VM configuration
        """Amount of parallel request. Related to cores per VM"""
        self.vm_parallel = 4
        self.vm_scaling_degradation = 0.00  # 5 percent degradation -> 0.05
        self.vm_min_instances = 0
        self.vm_auto_scaling = True

        # FaaS
        """Performance factor y = a + m * log(<memory in MB>)"""
        self.faas_performance_a = 49.61
        self.faas_performance_m = -6.49
```

Listing 5.2: Configuration for the instances and performance comparison of the simulation.

```
        # Request configuration
        self.request_duration = 2.055  # in seconds
        self.request_memory = 112  # in MB

        # random request generator
        self.request_generator_seed = 10  # seed for the random request generator
        self.request_variation = 0.0  # Percentage variation
```

Listing 5.3: Configuration for the requests of the simulation.

Following, the requests can be configured as shown in Listing 5.3. First, the `request_duration` and `request_memory` per request is specified (REQ-5). Additionally, the requests can be randomized by using `request_variation`. The randomness is reproducible (`request_generator_seed`), but changeable (REQ-6). The idea behind the randomization is to remove the fixed setting by introducing noise and creating a simulation closer related to the real world.

Then, the load is specified in Listing 5.4. Similar to the requests, also the load can be varied randomly. More information about the parameters is given in Section 5.3.3.

```
        # load generator
        self.load_generator_seed = 10
        self.load_generator_variation = 1
        self.load_generator_non_zeros = True

        # load
        self.load_name = None
        self.load_altitude = 1
        self.load_spacing = 3600
        self.load_num_requests = 10800
        self.load_num_spikes = 1
```

Listing 5.4: Configuration for the load of the simulation.

Finally, some parameters for caching (REQ-8) and plotting (REQ-9) are shown in Listing 5.5.

```
        # persist
        self.archive_folder = 'archive/'
        self.plotting_folder = 'images/'
        self.plotting = False
```

Listing 5.5: Configuration for caching and plotting of the simulation.

Pricing Information

The pricing information is provided via the `prices.json` file. In contains the instance types and price information used for the simulation. An external file is used to allow easy modification without changing any code (REQ-7).

Caching

As the execution of a simulation can take a while, the results of the simulation are cached. The location of the cache archive is shown in Section 5.3.3. Over time, the disk usage by the caches can grow significantly, therefore they should be emptied in case disk space becomes a scarce resource.

The simulator caches using Python binary objects through the `pickle`[3] library. Additionally, the web-interface also caches in the result in the json format. This is done, as the data transformation from Python object to json is not for free.

[3] `https://docs.python.org/3.7/library/pickle.html`

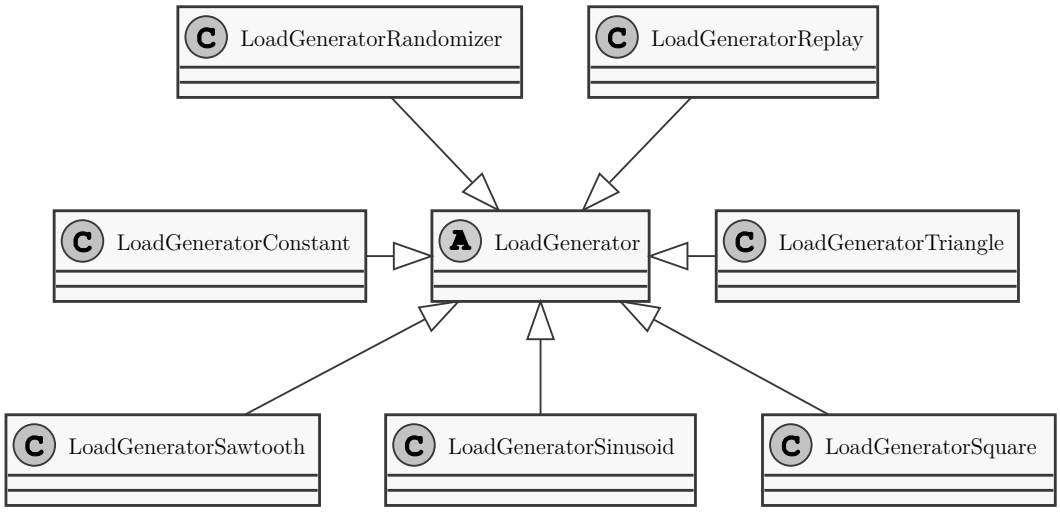

Figure 5.5.: Overview of the load patterns implemented in the simulator

Load Patterns

The load patterns themselves are explained in Section 5.2.2. They all implement the abstract class `LoadGenerator` as shown in Figure 5.5.

Additionally, `LoadGeneratorRandomizer` and `LoadGeneratorReplay` are added to the previous introduced load patterns. The `LoadGeneratorRandomizer` takes as input another `LoadGenerator` and randomizes the load according to the configuration as mentioned in Section 5.3.3. Moreover, the `LoadGeneratorReplay` uses as input a previously recorded load and reproduces exactly the same load again. The addition of more load patterns is possible as required by REQ-4.

Plotting

Plotting of the simulation results is included (REQ-9) and can be enabled as shown in Section 5.3.3. This is useful, if multiple simulations should be run, where the parameters are known and comparing of the image files is the preferred choice.

Web-Interface

A screenshot of the web-interface of the simulator is shown in Figure 5.6 (REQ-10). The view can be split into four components.

First, the top right graph shows the load as incoming requests per simulation step. This particular load is characteristic for a load following a stretched out normal distribution.

Second, the top left graph shows the cumulative cost over the time (red line) of the time of the simulation on the first y-axis. The secondary y-axis shows the cumulative total of provisioned instances required to handle the load.

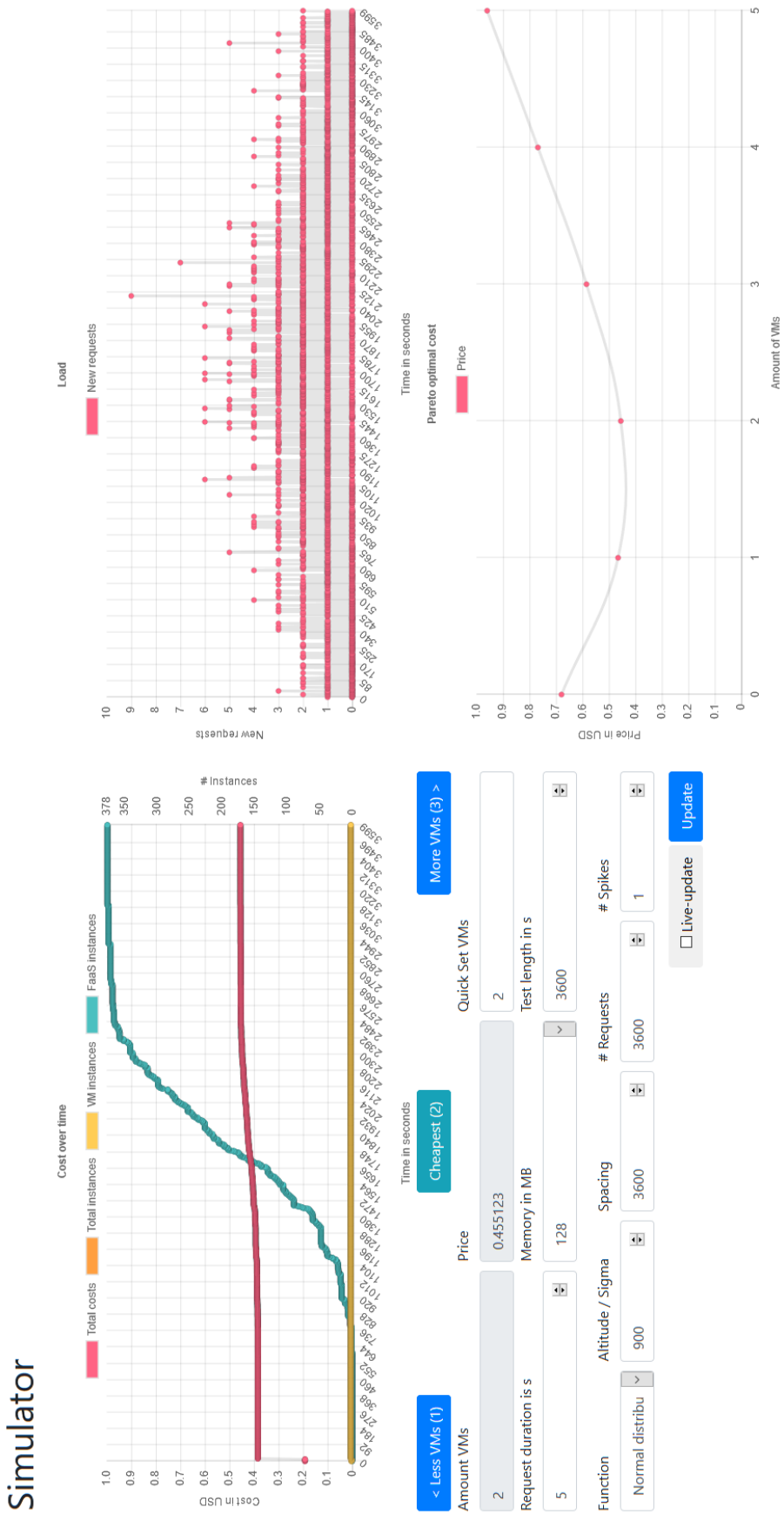

Figure 5.6.: Screenshot of the web-interface of the simulator

Third, the bottom right shows the Pareto front based on the number of provisioned VMs. The curve is lowest at $x = 2$ indicating that the most cost-efficient deployment strategy is serving the load with 2 VMs and the rest with FaaS.

Fourth, the bottom left is the configuration and information panel. The top shows the controls for interacting with the top left graph to change the number of used VMs. Also, a quick button to jump to the cheapest configuration is present. The second row shows information on the number of used VMs and associated price. Moreover, the number of VMs can be set directly. The remaining two rows allow for the configuration of requests and load characteristics. The *# Requests* fulfills a double functionality as for the normal distribution, the total number of requests can be set and for the other distributions the total number requests is shown and updated (read-only). Although the simulation is updated with any change in values immediately, it can be temporarily disabled via the *Live-Update* button if you want to change multiple values at once and then manually *Update*.

One note about the displayed data: The load of 3.600 requests (duration: 5 seconds, maximum memory: 128 MB) over the course of 1 hour does not seem particular bursty. Still, the optimal deployment strategy is a hybrid approach using FaaS and VMs.

5.4 Decision Model Configuration

The implemented decision model contains extra parameters compared to the original theoretical decision model, as discussed in Section 5.3.3.

Most parameters of the decision model are related to the specific use-case, which includes the load pattern, the duration of the requests, etc. Others describe the performance of cloud instances.

The AWS cloud is the most mature, as determined by the benchmark in Chapter 3. Therefore, the AWS cloud is selected as the cloud service provider for a custom cloud application, although any other cloud service provider also could have been chosen.

Furthermore, FFT with the input parameter $k = 131.072$ is picked for its short-duration, computational heavy calculations. The selected function is regarded as a typical request of the cloud application.

This selection is not suitable for every cloud application. In those cases, the following described experiment has to be repeated with the respective adjusted parameters.

5.4.1 Obtaining the Configuration Values

This section explains how to obtain the configuration values for *faas_performance*, *vm_parallel* and *vm_scaling_degradation* previously mentioned in Section 5.3.3.

In general, it is recommended to run the steps in an environment as equivalent or similar to production to gain meaningful results. This also includes using a functioning load-balancer to distribute the requests to multiple VMs-based instances.

For the first step, the scaling of the VMs-based instances is of interest. Secondly, the performance comparison to FaaS. Especially for the performance comparison, the same function or executable should be run. Also, caching e.g. by a cache proxy in any form needs to be disabled.

As stated before, FFT is used. For comparability to the results from the benchmark in Chapter 3, the original benchmark gets extend by a standalone version. It is essentially a Node.js server that has the same cloud functions as the benchmark, while supporting multiple requests at once. An excerpt from the code is shown in appendix B.1.

Scaling

The aspect scaling involves answering two questions:

- How many requests can one VM-based instance handle in parallel without performance degradation?
- What is the performance degradation per additional VM-based instance i.e. due to scaling limitations, etc.?

In both cases, the software JMeter[4] is used. It allows for easy scripting of so-called scenarios to performance test a software web application with multiple users in parallel. Each user continuously triggers the same cloud function for this test.

To answer the questions, the response time of the application is monitored. While slowly increasing the number of users and with it the number of concurrent requests, an (sudden) increase in response times indicate that the VM-based instances have reached their limits.

Running this with only one VM-based instance, leads to the answer of the first question. Secondly, by adding more VM-based instances more data points are collected to measure a possible performance degradation.

As a reference VM in the AWS cloud, the instance type *m5.xlarge* is chosen due to being advertised as a "General purpose instance" and a "good choice for many applications" (Amazon Web Services, Inc., 2018b).

[4] `https://jmeter.apache.org/`

Instances	Users	Throughput	$\frac{Throughput}{instance}$	Response Time avg (s)
1	4	2,22	2,22	2,250
2	8	4,35	2,17	5,445
3	12	6,43	2,14	1,860
4	16	8,39	2,09	1,895
5	24	10,76	2,15	2,216
Average:			2,15	2,055[5]

Table 5.1.: Measured throughput and response time in seconds on AWS EC2 with the *m5.xlarge* instance type

	128 MB	256 MB	512 MB	1024 MB	2048 MB
Measured Duration	43,369s	21,235s	16,849s	9,248s	3,144s
Trend Line Duration	37,259s	28,016s	18,773s	9,530s	0,287s

Table 5.2.: Measured and trend line duration of FFT with $k = 131.072$ for different memory configurations on AWS Lambda

The *m5.xlarge* instance (4 vCPUs, 16 GB memory) can handle 4 requests in parallel ($vm\_parallel = 4$). Further, Table 5.1 shows the results. The throughput lies around 2,15 (average), which is constant, so $vm\_scaling\_degradation = 0$.

Performance Comparison

Secondly, the performance comparison requires an answer to:

- By which factor does the performance of VM-based instances and FaaS differ?

The *faas_performance_a* and *faas_performance_m* parameters are used for the performance log function. The average request duration is known for the VM-based instances through the previous step to be $2{,}055s$.

From the background in Chapter 3 the average execution durations of FFT with the parameter $k = 131.072$ for different memory configurations is known and displayed in Table 5.2.

Based on these, a trend line is computed since on AWS Lambda CPU cycles are proportional to the available memory (BH-2). The values are determined via the R script shown in appendix B.2 to be $y = 131{,}961 + (-13{,}335) * log(memory)$. Adjusted to become a factor by dividing by $2{,}055s$, so $faas\_performance\_a = 49{,}61$ and $faas\_performance\_m = -6{,}49$.

Using this function, the trend line duration is calculated. The measured and trend line values differ to a variable degree, but it is a valid approximation when the cloud service providers link the CPU cycles to the allocated memory (BH-2).

[5]Excluding the 5,445 ms entry, as it is an outlier compared to the other data points.

5.5 Results

For this, the randomization of the load and duration of the requests is disabled to ease the reproductivity of this work and simplify the discussion. The parameters obtained in Section 5.4.1 are used as listed in Table B.1.

Figure 5.7 provides a first overview of the Pareto front for the different load patterns. The load patterns have to handle each the same number of 115.200 requests (altitude is 64, 32 for constant load). It reveals the different categories of the load patterns.

5.5.1 Constant and Squared Load

The constant pattern as well as the square produce constant load, although their Pareto optimal deployment strategies are differing greatly.

As expected, the constant load utilizes the VMs-based instances fully and thus the Pareto front follow a straight line. The best price is reached when an all VMs-based deployment strategy is used.

The load of the square pattern follows also a straight line, although the slope is different. Due to the requests being focused one only the first half of the monitored duration, twice as many VMs-based instances are required to handle the load. Interestingly enough, handling the load with only FaaS-based instances is the most cost-efficient.

5.5.2 Periodic Load

Surprising, periodic load based on the *sawtooth*, *sinusoid* and *triangle* load patterns have a similar cost whether being served using a full FaaS-based or VM-based deployments. It seems that the cost differences (shown in Figure 2.6) and utilization efficiency of the VM-based instances even out for the chosen input values to the decision model. A change in the total amount of requests only scales the graphs in the figure and has no impact on the shape of the graphs.

However, it is no surprise as Figure 5.7 shows, that the sinusoid load distribution, which is flatter and less bursty cannot achieve an as great cost reduction as the square and triangle load pattern. As the data shows in Figure 5.7, the Pareto front is a curve with the best deployment strategy being a hybrid one.

When deploying x-many VM-based instances, redirecting excess load to FaaS is not problem. FaaS instances scale without any further provisioning or configuration and request routing is possible due to DMAS-5.

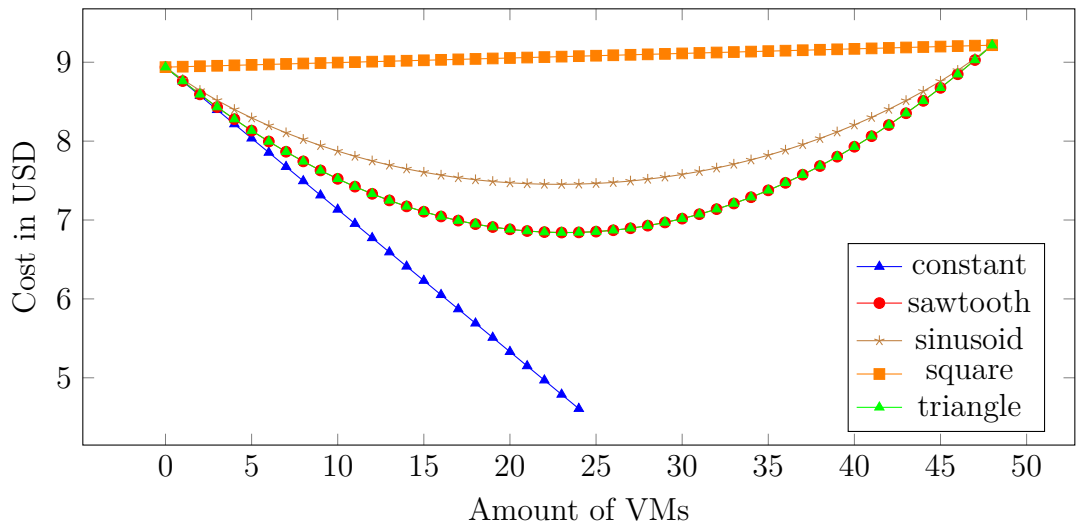

Figure 5.7.: Overview of the Pareto front for the different load patterns

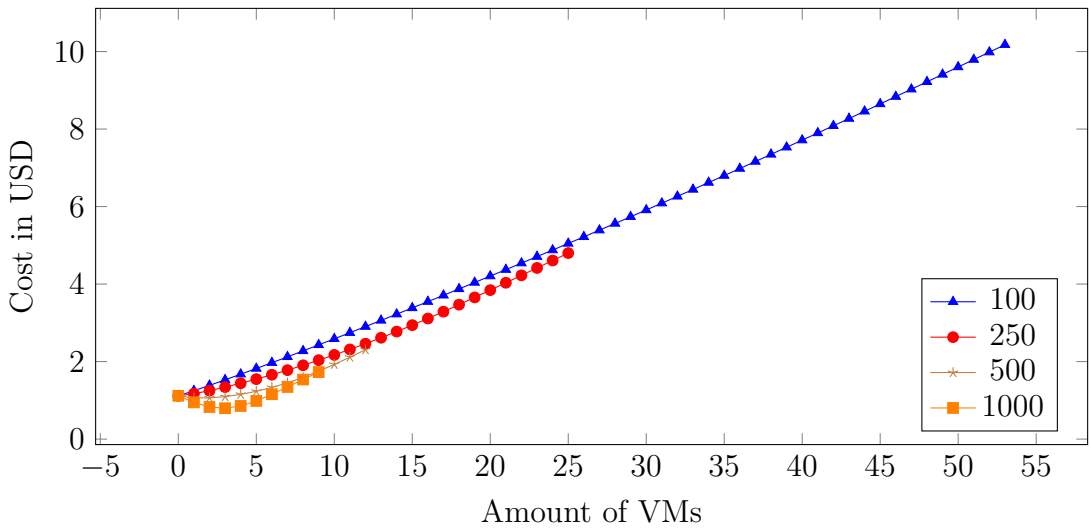

Figure 5.8.: Pareto front for loads following the normal distribution with different sigma values (8x less requests compared to Figure 5.7)

5.5.3 Bursty Load

Handling of load is more challenging during peak times as in a short interval a significant increase in load is measurable. As Eivy (2017) notices, even if the number of requests is constant over a time interval, the distribution makes the difference. Constant load can easily be handled, but sudden, unexpected peaks let an application become unresponsive or even go down.

To simulate different distributions of the same number of requests, the normal distribution with different sigma values is used. The burstiness is visualized in Figure 5.3, resulting in the Pareto front shown in Figure 5.8. Less requests (8x) are simulated, as an adjustment, the y-axis values can be multiplied by 8 to gain comparable values to Figure 5.7.

	FaaS		Hybrid		VM-based	
Load Distribution	**#VMs**	**Cost**	**#VMs**	**Cost**	**#VMs**	**Cost**
Constant	0	8,94	24	4,61	24	4,61
Square	0	8,94	0	8,94	48	9,22
Sawtooth	0	8,94	23	6,84	48	9,22
Sinusoid	0	8,94	23	7,45	48	9,22
Triangle	0	8,94	23	6,84	48	9,22
Normal Distribution (sigma=1.000)	0	8,94	24	6,37	72	13,82
Normal Distribution (sigma=100)	0	8,94	0	8,94	424	81,41

Table 5.3.: Cost comparison of a full FaaS, hybrid and full VM-based deployment solutions in USD on AWS Lambda for the different load distributions with 115.200 requests in total

The curves in Figure 5.8 show that in contrast to the other load patterns, FaaS-based instances are the choice for the most cost-efficient deployment strategy when handling bursty load.

Only as the sigma values gets large enough that the load becomes similar to a constant load, an all FaaS-based deployment strategy is not advisable[6]. Still, the Pareto front does not follow a straight line or favors an all VM-based deployment strategy. Remember, the sampling is done for the normal distribution introducing a natural non-continuous randomization.

5.5.4 Cost-Efficiency

Based on the previously shown Pareto fronts, a cost comparison between the different deployment strategies is possible. The cost overview is shown in Table 5.3. The normal distribution is adjusted to serve the same amount of load by multiplying its values by the factor 8.

The hybrid deployment strategy always finds the most cost-efficient deployment strategy. As already stated in the previous sections, an VM-based approach is best for constant load, an FaaS-based approach is best for bursty and periodic loads, a hybrid deployment model is advisable for spiky loads.

The hybrid deployment strategy does not improve the total cost for the constant load, but it does so for all other load patterns. The reduced cost in percentage is shown in Table 5.4.

[6]This is not really clear in the figure due to the choice of shape for sigma=10000

Load Distribution	FaaS	Hybrid
Constant	-93,96	0,00
Square	3,02	3,02
Sawtooth	3,02	25,77
Sinusoid	3,02	19,14
Triangle	3,02	25,77
Normal Distribution (sigma=1.000)	35,35	53,92
Normal Distribution (sigma=100)	89,02	89,02

Table 5.4.: Cost improvement in percentage compared to an all VM-based deployment strategy for the different load distributions with 115.200 requests in total

The shown cost and cost differences using the load defined by the parameters in Table B.1 consider one hour (one VM BTU on AWS, 32 requests per second, 115.200 requests in total) of operation. However, the percentage stay identical whether seen per hour or year.

Already for the periodic loads, cost savings around $\sim$ 25% are possible using a hybrid deployment strategy. Even the complete shift to a serverless deployment reduces the cost by $\sim$ 3%, which handles additional load without extra deployment effort.

In particular, bursty load benefits from FaaS and hybrid deployment strategies depending on the burstiness (defined by the sigma value). Besides the fact that the load distribution generated by the normal distribution resembles the reality closer as the others through the approach of sampling as seen in Section 5.3, even a minor peak is handled in a hybrid deployment strategy significantly more cost-efficient.

5.6 Conclusion

The implementation of the Decision Model in form of a simulator demonstrates the advantages of serverless computing. The results prove the expectations that especially non-constant load distributions benefit from serverless deployment models.

The cost reduction of any cloud application mainly depends on the load distribution. As it is shown in Section 5.5, cost saving for period loads are around $\sim$ 25% using a hybrid deployment strategy. Cloud applications with a bursty load can reduce cost even more, while being able to handle the load without delays or service outages.

The developed simulator is a first tool providing information on an optimal deployment strategy, using real benchmark data gathered in Chapter 3 as recommended by Kuhlenkamp and Klems (2017). The simulator is configurable to match the needs of any cloud application and incoming load. Additionally, it can reflect changes in the pricing and service models of the cloud service providers. Moreover, it can be extended as well.

For first observations how the load distribution has an effect on the optimal deployment strategy, the web-interface provides an easy way of interaction.

Chapter 6

Conclusion

Serverless computing is getting more attention and picking up speed. Therefore, it become inevitable to examine the field in an academic setting. Research exists but is still limited due to how recent the developments are.

The event-driven execution model is particularly relevant for many cloud applications. Additionally, automatic scaling is handled by the cloud service provider, while only the actual resource consumption is billed.

6.1 Summary

The first research question of this work addresses the performance of public FaaS solutions. For answering the question, a benchmark is developed and executed. The findings are surprising as the FaaS solutions differ not only on their technical specifications, but in many aspects including performance and total cost. The field of serverless computing is still in development and so are the FaaS implementation by the benchmarked cloud service providers: AWS Lambda, Google Cloud Functions, IBM Cloud Functions and IBM Cloud Functions. This is also partly covered in Back and Andrikopoulos (2018), which builds upon this work.

The second research question builds on top of the lessons learned from the benchmark. The goal is to find the optimal hybrid deployment strategy for any cloud application. Therefore, first a generic decision model is introduced to find deployment strategies and to choose the optimal one in terms of cost-efficiency.

The decision model is implemented in form of a simulator. The simulator offers tooling support for consumers to make educated decisions about the most cost-efficient deployment strategy for their specific cloud application. The web-interface allows easy interaction and fast scenario simulations.

Eivy (2017) writes: “The economic benefits of serverless computing heavily depend on the execution behavior and volumes of the application workloads”. The constant load is still best handled using VM-based deployment solutions as Albuquerque Jr et al. (2017) claim. They are also correct, that in particular cloud applications with a irregular load benefit from serverless computing as this is shown with the decision model.

Moreover, cloud applications with period load and a hybrid deployment strategy, as described in the decision model and applied through the simulator, can reduce their cost by $\sim 25\%$.

Especially bursty load can be handled well with serverless computing by scaling the cloud application without service outage due to too high usage. Assuming that the cloud application was previously able to handle the load without a delay by over-provisioning of resources, an even higher cost reduction can be achievable. In essence, over-provisioning is not required anymore, as serverless allows for resources provisioning and consumption on-demand in milliseconds.

It is also shown, that the best deployment strategy and performance depends heavily on the specific cloud application. Therefore, the parameters for the decision model and simulator are customizable, so that any cloud application and upcoming changes on side of the cloud service providers can be incorporated.

Depending on the popularity and billed cost of a cloud application, the cost saving can even justify refactoring the application into the serverless model.

6.2 Future Work

During the execution of this work, limitations were identified and new questions have risen. To move forward, these need to be addressed in the future work.

A re-run of the benchmark to gain a better insight on provider performance in the depth of time is needed. Especially the production-ready version of Google Cloud Functions needs to be examined. A more stable and improved FaaS implementation is expected. Additionally, suggestions for a better benchmark are listed in Section 3.5.

One of the original ideas in finding an optimal hybrid deployment strategy was the use of a MAPE-K loop with a rule-engine. The advantage lies in expressing only rules for which the algorithm then automatically optimizes. At the current moment, the multi-dimensionality and with it the complexity of the hybrid model poses a problem. The advantage of the rule-engine is the better expressiveness while less assumptions are needed required compared to the presented approach.

Deciding if a change in load is due to a sudden peak or a long-term change depends on the application and requires probably a human in the loop. An adjustment of the deployment strategy might be necessary. Therefore, learning based on previously recorded load patterns or changes in the last BTU is an interesting addition. For classical clouds and in combination with MAPE-K, Jamshidi et al. (2016) have produced promising results.

This work builds on top of the assumption of a perfect load-balancer. While an almost perfect one exists for homogeneous load, heterogeneous load triggering different cloud functions with different resource requirements cannot be handled perfectly at the moment. Further research is necessary to find an ideal solution for implementation.

Another outlook goes towards hybrid clouds, where private and public clouds get combined. This work has looked at the minimization of cost of different cloud computing deployment models within one cloud. The research should be expanded to mixing multiple clouds together. The challenges lie in the latency and possible split up into two distinct clouds due to networking problems.

Also the aspects of software engineering like modularity, testing, reliability and life cycle management as mentioned in van Eyk et al. (2017) are important and so is security. Besides the usual risks, the automatic scaling by concurrent executions can lead to high bills and be actively abused by attackers as stated by Eivy (2017).

In the future, the serverless paradigm can bring back flow-based programming in a cloud-native programming model as seen as a new perspective by van Eyk et al. (2017). The idea of processing data in data streams according to a work-flow fits the fine grain responsibility of the stateless cloud functions.

Chapter A

Benchmark

A.1 Execution Script

Executing the benchmark is done by the script in Listing A.1. The file is presented in summary here, by removing some functions as well as the `servers` variable to increase readability.

The benchmark script uses the `curl` command to query the cloud service providers.

```
#!/bin/bash
function get_curl_params {
  url=$1
  param=$2
  if [[ "$url" =~ "api/v1/namespaces" ]]; then
    # found OpenWhisk installation. Add curl parameters (+auth)
    echo -ne -X POST --data "{\"param\":$param}" -k -u $OpenWhiskAuth -H ↩
     Content-Type:application/json
  else
    echo -ne -X GET
  fi
}

function run_function {
  server=$1
  function=$2
  params=$3

  echo -ne "\nRun $function..."

  # Call three times before to warm up
  URL="$server$function?param=0&blocking=true"
  curlParams=$(get_curl_params $URL "0")
  echo -ne " (warming up: "
  curl $curlParams -s $URL
  echo -ne "."
  curl $curlParams -s -o /dev/null $URL
  echo -ne "."
  curl $curlParams -s -o /dev/null $URL
```

```
  echo " done)"

  # test all parameters
  for param in "${params[@]}"
  do
    echo -ne "Run $function (param=$param)..."
    URL="$server$function?param=$param&blocking=true"
    curlParams=$(get_curl_params $URL $param)
    result=$(curl ${curlParams} -s "$URL")
    echo "$result"

    echo "$(date -u +%Y-%m-%dT%H:%M:%SZ), $function, $param, $URL, $result"↩
      >> getRequest.csv.log
  done
}


# The execution starts here
echo -e "This runs the benchmark..."

for server in "${servers[@]}"
do
  echo
  echo "#######################################"
  echo $server
  echo "#######################################"

  # FFT
  functions=( fft128 fft256 fft512 fft1024 fft2048 )
  params=(8192 16384 32768 65536 131072 262144 524288 1048576 2097152 4194304 ↩
    8388608 16777216)
  for function in "${functions[@]}"
  do
    run_function $server $function $params
  done
done
```

Listing A.1: Script to execute the benchmark

A.2 Serverless Wrapper Code

For each cloud service provider's FaaS implementation, the same code is run. However, the cloud service provider's FaaS interface differ. Therefore, a custom minimal wrapper was written. The code in Listing A.2 shows part of the wrapper code for the Google Cloud Functions platform.

```
'use strict';

// File for Google Functions

const Fft = require("./code/fft.js");
const Fibonacci = require("./code/fibonacci.js");
const Matrix = require("./code/matrix.js");
const Pi = require("./code/pi.js");
const Sleep = require("./code/sleep.js");
const UnionFind = require("./code/unionfind.js");


const fftFunc = (request, response) => {
  const param = request.query.param || 0;
  console.log('Input param=' + param);

  const result = Fft.fft(param);

  response.status(200).send(JSON.stringify(result));
};
module.exports.fft128 = fftFunc;
module.exports.fft256 = fftFunc;
module.exports.fft512 = fftFunc;
module.exports.fft1024 = fftFunc;
module.exports.fft2048 = fftFunc;
```

Listing A.2: Excerpt of the wrapper code to run the cloud functions on Google Cloud Functions

A.3 BH-2 Speed-ups between Memory Configurations

Table A.1 and Table A.2 show the speed-ups of the cloud functions on AWS Lambda and Google Cloud Functions respectively. From 128 MB to 512 MB, an increase of 4x is expected. A 16x increase is expected for 128 MB to 2048 MB on AWS Lambda, while 12x for Google Cloud Functions due to different CPU provisioning (see Table 2.1).

Function	128 MB to 512 MB	128 MB to 2048 MB
FFT	2,49	13,58
MM	1,05	3,19
PI	3,26	14,97
S	1,00	1,00
UF	4,20	13,31

Table A.1.: Speed-up of the cloud functions on AWS Lambda over different memory configurations

Function	128 MB to 512 MB	128 MB to 2048 MB
FFT	4,81	9,64
MM	1,05	2,14
PI	3,12	6,03
S	1,06	1,06
UF	4,09	8,57

Table A.2.: Speed-up of the cloud functions on Google Cloud Functions over different memory configurations

S has no speed-up as expected and surprisingly the speed-up of MM is low. Therefore, not all cloud functions benefit as much from a higher performing CPU.

The cloud functions FFT, PI and UF have an acceptable speed-up, although stay below their possibilities. The reason is not known at this point in time and requires further investigation.

A.4 BH-4: The billed duration matches the execution time

Figure A.1 shows the measured durations of S128, in opposite to Figure 3.5 showing S2048. Essentially, the results are identical. BH-2 has no impact on the precision as previously assumed by BH-4.

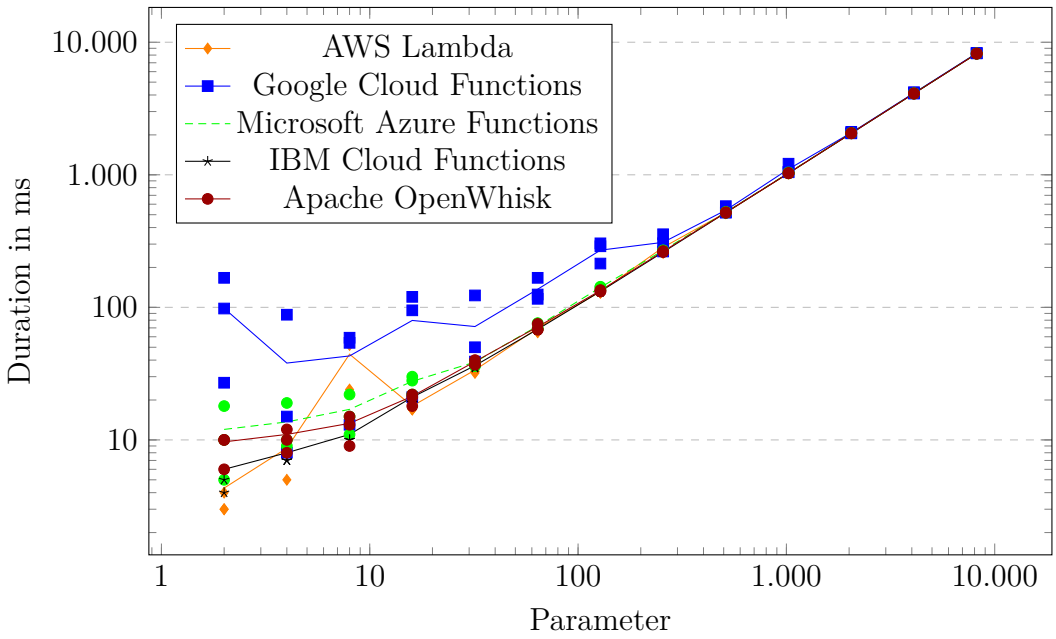

Figure A.1.: Measured successful durations of S128 across all cloud service provider

A.5 Memory Consumption

The following tables show the used memory on the AWS Lambda platform for the different cloud functions and their input parameters. AWS Lambda is the only platform reporting the peak memory consumption. The maximum value out of all measurements is shown.

For S, the memory allocation is constant as expected. The consumption lies between 28 and 30 MB. This is not the minimum memory usage as the dependencies for the other algorithms used in this benchmark are loaded as well.

Parameter	8.192	16.384	32.768	65.536	131.072	262.144	524.288	1.048.576	2.097.152	4.194.304	8.388.608	16.777.216
Memory	33	44	59	81	112	200	251	474	848	1.840	-	-

Table A.3.: Measured memory consumption of FFT in MB on AWS Lambda using the maximum value

Parameter	100	200	300	400	500	600	700	800	900	1.000	1.500
Memory	28	29	33	40	48	63	70	118	146	150	182

Table A.4.: Measured memory consumption of MM in MB on AWS Lambda using the maximum value

Parameter	64	128	256	512	1.024	2.048	4.096	8.192	16.384	32.768	65.536	131.072	262.144	524.288	1.048.576
Memory	35	36	37	37	44	46	46	46	46	47	47	47	63	64	78

Table A.5.: Measured memory consumption of PI in MB on AWS Lambda using the maximum value

Parameter	65.536	131.072	262.144	524.288	1.048.576	2.097.152	4.194.304	8.388.608	16.777.216	33.554.432
Memory	40	62	109	169	335	619	1.157	1.600	2.047	-

Table A.6.: Measured memory consumption of UF in MB on AWS Lambda using the maximum value

Chapter B

Simulator

This chapter contains data, code, observations and other findings related to the Simulator.

B.1 Standalone Version of the Cloud Functions

The following code is used to run the standalone version of the benchmark. It is a Node.js application wrapper to use the same function as for the FaaS Benchmark.

The standalone version is a wrapper around the code of Google Cloud Functions deployment. Additionally, a cluster mode to process multiple requests concurrently is added.

```
'use strict';

// Standalone server

const cluster = require('cluster');
const http = require('http');
const url = require('url');

// index.js contains the google FaaS implementation
const functions = require('./index.js');
const numCPUs = require('os').cpus().length;
const port = process.env.PORT || 8000;

function processRequest(request, response) {
  const parsed = url.parse(request.url, true)
  console.log(parsed.query);

  // patching the environment to become like google
  request.query = parsed.query;
  response.status = function (status) {
    response.writeHead(status);
    return response;
  }
```

```
24    response.send = function (data) {
25      response.write(data);
26      response.end();
27      return response;
28    }
29
30    try {
31      // find the right function to execute
32      const requestUrl = parsed.pathname.substr(1);
33      switch (requestUrl) {
34        case 'fft128':
35          functions.fft128(request, response);
36          break
```

⋮

```
136         default:
137           response.status(204).send('Unknown function');
138
139       }
140
141       console.log(requestUrl);
142     } catch (e) {
143       response.writeHead(500)
144       response.end();
145       console.log(e.stack);
146     }
147   }
148
149
150   // Start the HTTP server
151   if (cluster.isMaster) {
152     // Use a cluster to benefit from multi-core machines
153     console.log(`Master ${process.pid} is running`);
154
155     // Fork workers.
156     for (let i = 0; i < numCPUs; i++) {
157       cluster.fork();
158     }
159
160     cluster.on('exit', (worker, code, signal) => {
161       console.log(`worker ${worker.process.pid} died`);
162       cluster.fork();
163     });
164   } else {
165     // Workers can share any TCP connection
166     http.createServer(processRequest).listen(port);
167
168     console.log(`listening on port ${port}`);
169   }
```

Listing B.1: Standalone wrapper code with concurrent request processing support

B.2 Code for Finding the Trend Line

The following code is used to calculate the trend line parameters. The used values are embedded in the code.

```
# Taken from https://stackoverflow.com/questions/15102254/how-do-i-add-different-trend-lines-in-r#15102307
# Adjusted the input x and y values


# set the margins
tmpmar <- par("mar")
tmpmar[3] <- 0.5
par(mar=tmpmar)

# get underlying plot
x <- c(128, 256, 512, 1024, 2048)
y <- c(43.369, 21.235, 16.84933333, 9.248, 3.144333333)
plot(x, y, pch=20)

# basic straight line of fit
fit <- glm(y~x)
co <- coef(fit)
abline( fit , col="blue", lwd=2)
paste('y =', co [[2]], '* x', '+', co [[1]])

# exponential
#f <- function(x,a,b) {a * exp(b * x)}
#fit <- nls(y ~ f(x,a,b), start = c(a=1, b=1))
#co <- coef(fit)
#curve(f(x, a=co[1], b=co[2]), add = TRUE, col="green", lwd=2)

# logarithmic
f <- function(x,a,b) {a * log(x) + b}
fit <- nls(y ~ f(x,a,b), start = c(a=1, b=1))
co <- coef(fit)
curve(f(x, a=co[1], b=co[2]), add = TRUE, col="orange", lwd=2)
paste('y =', co [[1]], '* log(x)', '+', co [[2]])

# polynomial
f <- function(x,a,b,d) {(a*x^2) + (b*x) + d}
fit <- nls(y ~ f(x,a,b,d), start = c(a=1, b=1, d=1))
co <- coef(fit)
curve(f(x, a=co[1], b=co[2], d=co[3]), add = TRUE, col="pink", lwd=2)
paste('y =', co [[1]], '* x^2', '+', co [[2]], '* x +', co [[3]])
```

Listing B.2: Calculating a trend line in R

B.3 Evaluation

Additional information and visualizations related to the Results in Section 5.5 are provided in this section.

B.3.1 Parameters

The parameters used are listed in Table B.1. The way of determining them is explained in Section 5.4.1.

Parameter	Value
`simulation_start`	0
`simulation_end`	3.600
`vm_parallel`	4
`vm_scaling_degradation`	0
`vm_auto_scaling`	False
`faas_performance_a`	49,61
`faas_performance_m`	-6,49
`request_duration`	2,055
`request_memory`	112
`load_name`	{'constant', 'sawtooth', 'sinusoid', 'square', 'triangle'} or {'ndist'}
`load_altitude`	{1, 2, 4, 8, 16, 32, 64} or {100, 250, 500, 750, 1.000} (Normal Distribution)
`load_spacing`	3.600
`load_spikes`	1
`load_num_requests`	14.400

Table B.1.: Parameters used in the simulation

B.3.2 Constant and Squared Load

If the application has the constant load pattern, then an all VM-based solution is most cost-efficient. The altitude has no effect on the form of the Pareto optimal curve, only the y-axis scaling.

Notice the different y-axis scaling due to the constant pattern having for the same altitude, twice as many requests.

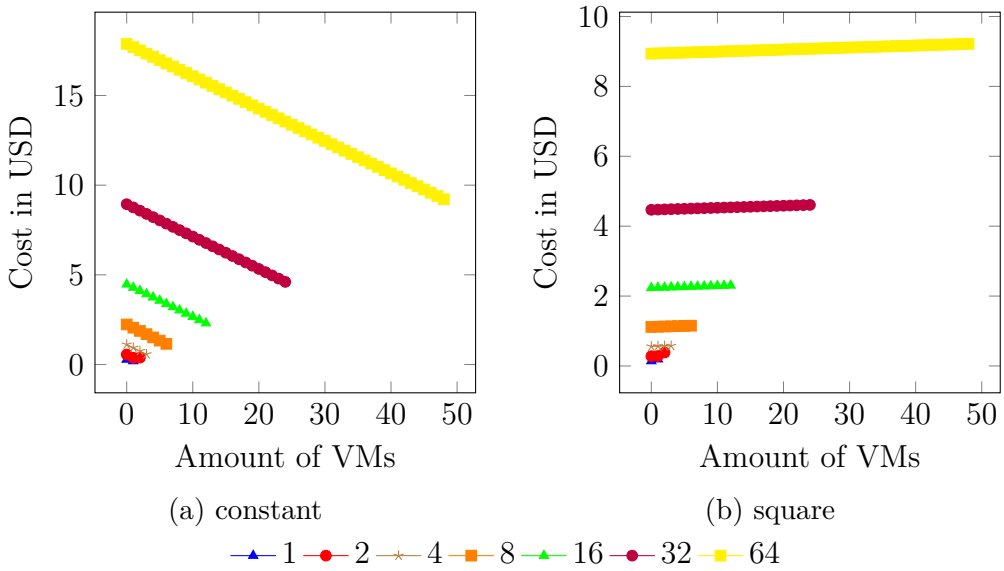

Figure B.1.: Pareto optimal deployment strategies for the constant and squared load pattern with varying altitudes

B.3.3 Periodic Load

If the application has a periodic load pattern, then a hybrid deployment strategy is most cost-efficient. That is shown in Figure B.2a, Figure B.2b and Figure B.2c.

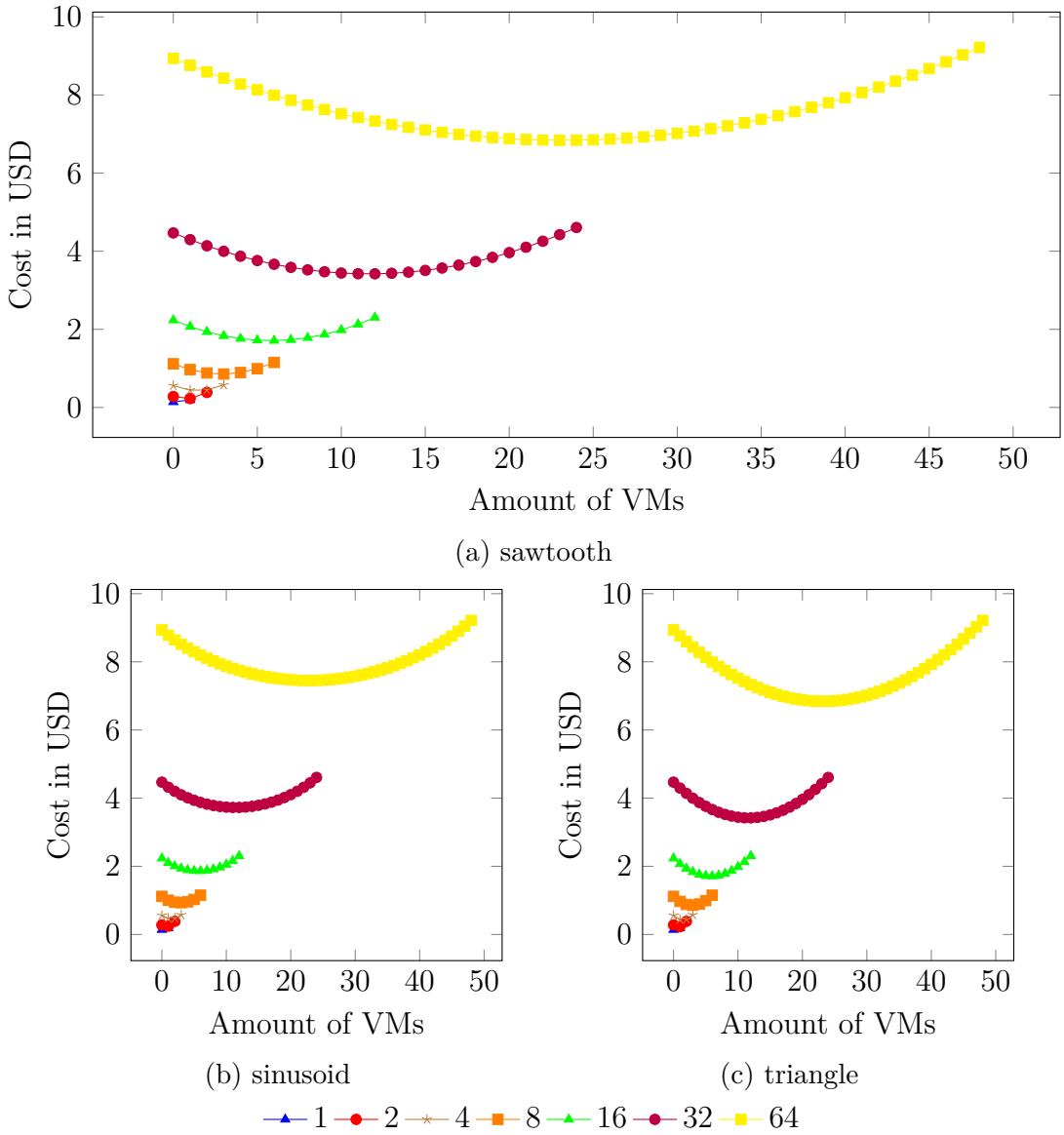

Figure B.2.: Pareto optimal deployment strategies for the periodic load patterns with varying altitudes

Bibliography

L. F. Albuquerque Jr, F. S. Ferraz, R. F. Oliveira, and S. M. Galdino. Function-as-a-service x platform-as-a-service: Towards a comparative study on faas and paas. *ICSEA 2017*, page 217, 2017.

Amazon Web Services, Inc. Aws lambda. `https://aws.amazon.com/lambda/`, 2018a.

Amazon Web Services, Inc. Aws ec2 instance types - amazon web services (aws). `https://aws.amazon.com/ec2/instance-types/`, 2018b.

Amazon Web Services, Inc. Aws lambda pricing. `https://aws.amazon.com/lambda/pricing/`, 2018c.

Apache OpenWhisk. Apache openwhisk is a serverless, open source cloud platform. `http://http://openwhisk.incubator.apache.org/`, 2018.

T. Back and V. Andrikopoulos. Using a microbenchmark to compare function as a service solutions. In *Service-Oriented and Cloud Computing*. Springer, Springer International Publishing, 2018.

L. Badger, T. Grance, R. Patt-Corner, and J. Voas. Cloud computing synopsis and recommendations. national institute of standards and technology (nist), special publication 800–146. us department of commerce. *US Department of Commerce. May*, pages 800–146, 2011.

I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

R. S. Barga. Serverless computing: Redefining the cloud, Jun 2017. URL `https://www.serverlesscomputing.org/wosc17/presentations/barga-keynote-serverless.pdf`.

A. Belloum. Cookery in aws lambda. Bachelor's thesis, University of Amsterdam, The Netherlands, 2017.

S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119 (Best Current Practice), Mar. 1997. URL `http://www.ietf.org/rfc/rfc2119.txt`.

R. Buyya, S. N. Srirama, G. Casale, R. Calheiros, Y. Simmhan, B. Varghese, E. Gelenbe, B. Javadi, L. M. Vaquero, M. A. Netto, et al. A manifesto for future generation cloud computing: Research directions for the next decade. *arXiv preprint arXiv:1711.09123*, 2017.

R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011.

T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. Computer science. MIT Press, 2009. ISBN 9780262533058.

Docker, Inc. What is a container — docker, Aug 2018. URL `https://www.docker.com/resources/what-container`.

A. Eivy. Be wary of the economics of "serverless" cloud computing. *IEEE Cloud Computing*, 4(2):6–12, 2017.

S. Frey and W. Hasselbring. The cloudmig approach: Model-based migration of software systems to cloud-optimized applications. *International Journal on Advances in Software*, 4(3 and 4):342–353, 2011.

S. Gómez Sáez, V. Andrikopoulos, M. Hahn, D. Karastoyanova, F. Leymann, M. Skouradaki, and K. Vukojevic-Haupt. Performance and cost evaluation for the migration of a scientific workflow infrastructure to the cloud. In *CLOSER*, pages 352–361, 2015.

S. Gómez Sáez, V. Andrikopoulos, M. Bitsaki, F. Leymann, and A. V. Hoorn. Utility-based decision making for migrating cloud-based applications. *ACM Transactions on Internet Technology (TOIT)*, 18(2):22, 2018.

Google LLC. Cloud functions - event-driven serverless computing — google cloud. `https://cloud.google.com/functions/`, 2018a.

Google LLC. Google functions - pricing — google cloud. `https://cloud.google.com/functions/pricing-summary/`, 2018b.

Google LLC. Google trends - serverless. `https://trends.google.com/trends/explore?date=all&q=serverless`, 2018c.

S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with OpenLambda. *Elastic*, 60:80, 2016.

IBM. Cloud functions overview — ibm cloud. `https://www.ibm.com/cloud/functions`, 2018.

P. Jamshidi, C. Pahl, and N. C. Mendonça. Managing uncertainty in autonomic cloud elasticity controllers. *IEEE Cloud Computing*, 3(3):50–60, 2016.

E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 445–451. ACM, 2017.

K. M. Khalil and Y. S. Sun. The effect of bursty traffic on the performance of local area networks. In *Proc. IEEE Global Telecommunications Conf. Conf. Record., GLOBECOM '92. Communication for Global Users*, pages 597–603 vol.1, Dec. 1992. doi: 10.1109/GLOCOM.1992.276445.

J. Kuhlenkamp and M. Klems. Costradamus: A cost-tracing system for cloud-based software services. In *International Conference on Service-Oriented Computing*, pages 657–672. Springer, 2017.

H. Lee, K. Satyam, and G. Fox. Evaluation of production serverless computing environments. Technical report, Indiana University Bloomington, 04 2018. URL `http://dx.doi.org/10.13140/RG.2.2.28642.84165`.

P. Leitner, J. Cito, and E. Stöckli. Modelling and managing deployment costs of microservice-based cloud applications. In *Proc. IEEE/ACM 9th Int. Conf. Utility and Cloud Computing (UCC)*, pages 165–174, Dec. 2016.

Z. Li, H. Zhang, L. O'Brien, R. Cai, and S. Flint. On evaluating commercial cloud services: A systematic review. *Journal of Systems and Software*, 86(9): 2371–2393, 2013.

W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara. Serverless computing: An investigation of factors influencing microservice performance. *IEEE International Conference on Cloud Engineering (IC2E)*, 2018.

M. Malawski, K. Figiela, A. Gajek, and A. Zima. Benchmarking heterogeneous cloud functions. In *Euro-Par 2017: Parallel Processing Workshop*, pages 415–426. Springer, 2017a.

M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela. Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions. *Future Generation Computer Systems*, 2017b.

D. C. Marinescu. *Cloud computing: theory and practice.* Morgan Kaufmann, 2017.

M. Maurer, I. Breskovic, V. C. Emeakaroha, and I. Brandic. Revealing the mape loop for the autonomic management of cloud infrastructures. In *Proc. IEEE Symp. Computers and Communications (ISCC)*, pages 147–152, June 2011. doi: 10.1109/ISCC.2011.5984008.

P. Mell and T. Grance. The nist definition of cloud computing, recommendations of the national institute of standards and technolog. *National Institute of Standards and Technology*, pages 800–145, 2011.

Merriam-Webster. Benchmark — definition of benchmark by merriam-webster, Jun 2018a. URL `https://www.merriam-webster.com/dictionary/benchmark`.

Merriam-Webster. Simulator — definition of simulator by merriam-webster, Jun 2018b. URL https://www.merriam-webster.com/dictionary/simulator.

Microsoft Corporation. Azure functions - serverless architecture — microsoft azure. https://azure.microsoft.com/en-us/services/functions/, 2018.

P. Müns. Serverless (faas) vs. containers - when to pick which?, Oct 2017. URL https://serverless.com/blog/serverless-faas-vs-containers/.

D. Montgomery and G. Runger. *Applied Statistics and Probability for Engineers*. John Wiley & Sons, 2010. ISBN 9780470053041.

F. F.-H. Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology*, 23(3):153–163, 2004. doi: 10.1080/01449290410001669914. URL https://doi.org/10.1080/01449290410001669914.

P. Pardalos, A. Migdalas, and L. Pitsoulis. *Pareto Optimality, Game Theory and Equilibria*. Springer Optimization and Its Applications. Springer New York, 2008. ISBN 9780387772479.

RightScale. RightScale 2018 State of the Cloud Report, 2018. URL https://www.rightscale.com/lp/state-of-the-cloud.

M. Roberts. Serverless architecture, Mar 2014. URL https://www.martinfowler.com/articles/serverless.html.

M. Roberts and J. Chapin. *What is Serverless?* O'Reilly Media, 2017.

U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. A cost-aware elasticity provisioning system for the cloud. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 559–570. IEEE, 2011.

J. Spillner. Snafu: Function-as-a-service (faas) runtime design and implementation. *ArXiv e-prints*, 2017a.

J. Spillner. Transformation of python applications into function-as-a-service deployments. *ArXiv e-prints*, 2017b.

J. Spillner. Exploiting the cloud control plane for fun and profit. *arXiv preprint arXiv:1701.05945*, 2017c. URL https://arxiv.org/pdf/1701.05945.pdf.

J. Spillner and S. Dorodko. Java code analysis and transformation into aws lambda functions. *ArXiv e-prints*, 2017.

J. Spillner, C. Mateos, and D. A. Monge. Faaster, better, cheaper: The prospect of serverless scientific computing and hpc. In *Latin American High Performance Computing Conference*, pages 154–168. Springer, 2017.

Statista. Annual revenue of amazon web services from 2013 to 2017 (in million u.s. dollars), 2018. URL https://www.statista.com/statistics/233725/development-of-amazon-web-services-revenue/.

E. van Eyk, A. Iosup, S. Seif, and M. Thömmes. The spec cloud group's research vision on faas and serverless architectures. In *Proceedings of the 2nd International Workshop on Serverless Computing*, pages 1–4. ACM, 2017.

M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference (10CCC), 2015 10th*, pages 583–590. IEEE, 2015.

M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang. Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In *Proc. Cloud and Grid Computing (CCGrid) 2016 16th IEEE/ACM Int. Symp. Cluster*, pages 179–182, May 2016. doi: 10.1109/CCGrid.2016.37.

M. Yan, P. Castro, P. Cheng, and V. Ishakian. Building a chatbot with serverless computing. In *Proceedings of the 1st International Workshop on Mashups of Things and APIs*, page 5. ACM, 2016.

Declaration of Authorship

I, Timon Back, declare that this thesis titled, "Hybrid serverless and virtual machine deployment model for cost minimization of cloud applications" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____