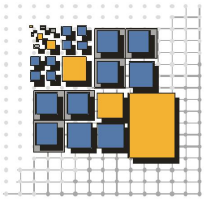


Otto-Friedrich-Universität Bamberg

Lehrstuhl für Praktische Informatik

Bachelorarbeit

im Studiengang Angewandte Informatik
der Fakultät Wirtschaftsinformatik und Angewandte Informatik
der Otto-Friedrich-Universität Bamberg

Zum Thema:

Containerized Profiling of Cloud Functions

Vorgelegt von:

Martin Endreß

Themensteller:
Prof. Dr. Guido Wirtz

Abgabedatum:
27.09.2019

Contents

List of Figures

List of Tables

Listings

Abbreviations

AWS	Amazon Web Service
CaaS	Container as a Service
cgroups	control groups
CSV	Comma Separated Values
DevOps	Development Operations
FaaS	Function as a Service
FLOPS	Floating Point Operations per Second
GFLOPS	$10^9 \times$ Floating Point Operations per Second
GCP	Google Cloud Platform
HMM	Hidden Markov Model
JVM	Java Virtual Machine
NoOps	No Operations
OOM	Out of Memory
QoS	Quality of Service
VM	Virtual Machine

Abstract

Serverless Computing has attracted a lot of attention in recent years. In the Function as a Service realisation of the serverless paradigm, the developer is released from operational concerns and only has to provide for the business logic of a service. However, in most cases, developers still face the challenge of choosing the amount of CPU and memory allocated for their cloud functions. Since it is not an easy, yet an important task, we provide a decision aid based on an offline, containerized profiling approach. With the goal of better understanding the resource usage of a cloud function, we run it locally inside a container and measure its resource demands. For this, we propose to map the execution environment of a cloud provider to the local machine. The aim is to get closer to the local simulation of the cloud environment and thereby provide meaningful information about a cloud function prior to its deployment. From there, we develop a model which describes the cloud function in a mostly hardware independent way.

1 Introduction

Serverless computing promises the abstraction from any operational aspect that previously was the responsibility of the developers, the operations team or a mixture in the form of a Development Operations (DevOps) team. The term "serverless" describes the perspective of the service consumer, who seemingly is exempt from operational concerns. Some blogs go as far as to say that serverless computing introduces No Operations (NoOps) as an advancement and replacement of DevOps[1]. They promote that developers only have to provide a piece of code that is then available anywhere at a specified endpoint. While the claim shows great promise, this abstraction is not complete as cloud service consumers are still responsible for a lot of management tasks regarding resource allocation as well as higher level concerns [JSSS+19, vEIST17, Eiv17]. In the Function as a Service (FaaS) implementation of the serverless paradigm they must define the provisioned CPU and memory resources, the deployment region, and security features, among other things. Additionally, on a higher level they need to establish versioning and update mechanisms, while also monitoring and debugging the architecture to find anomalies. Due to the heterogeneity of service models, the necessary responsibilities as well as the options offered to consumers may vary depending on the provider. Instead of the envisioned full abstraction from the server infrastructure, the term DevOps is yet to be defined in the serverless and especially in the FaaS context [vEIST17].

Cloud consumers are faced with the difficult task of assigning resources to their cloud functions. To aid in this decision making process, this thesis will investigate the resource demands of a cloud function. The goal is to evaluate the resource demands of the function independently of provider and local execution environments. This section will motivate and define the research questions as well as examine related work.

1.1 Motivation

The setup and maintenance of dedicated computer infrastructure represents a big cost factor especially for small to medium sized companies. In addition to fixed overhead costs that are independent of demand, companies run the risk of over- or under provisioning their hardware. As a solution, cloud computing offers several models which provide computing resources as a utility [AFG+10].

Because of the abstraction from all operational concerns and the seamless scaling property, the serverless paradigm and especially the FaaS model have grown in popularity. In the FaaS computing model, the developer uploads a function that implements the desired business logic in a programming language of their choice and selects an event that triggers its execution. Ideally, the function is then globally available at a specified endpoint without the need for further configuration. For this purpose, the tasks related to the operational logic like resource provisioning and autoscaling are completely taken over by the cloud provider [vEIST17]. This strict separation of business and operational logic requires that tasks associated with DevOps have to be clarified in the FaaS model [vEIST17].

Unfortunately, this separation has not yet been fully achieved, as recent literature de-

[1] Serverless Computing: Moving from DevOps to NoOps:
`https://devops.com/serverless-computing-moving-from-devops-to-noops/`

scribes technical difficulties in the realisation of the FaaS paradigm [JSSS+19, HFG+18]. Although the vast majority of the specification is the responsibility of the provider, the CPU and memory resource allocation of a cloud function must be set by the consumer most of the time. Service consumers are responsible for determining and selecting the optimal parameters for each of their cloud functions. Depending on their needs, they have to take into account performance, availability, and costs. Finding this local optimum is not an easy task. However, the choice entails significant impact on performance and cost, since resources used on function level are paid regardless of their utilisation. The choice of appropriate resource settings is therefore crucial. One option is to use the provider's monitoring information of previous function executions as a basis for this decision [JSSS+19, Eiv17]. In this scenario, a consumer would first deploy the function with an arbitrary setting and then iteratively adjust the settings based on the resource demands of the observed executions.

Eivy claims that the "real execution is the only valid test" of a function running in the cloud [Eiv17, p. 9]. They argue that because of the heterogeneity of the cloud, running the cloud function on a local machine will significantly impair the performance of the function. Specifically, its performance depends on the architecture used by a cloud provider and changes with capacity utilisation, resource contention through "noisy neighbours", and the varying hardware infrastructure used. Eivy's argument is unsatisfactory as it implies that a provider is already selected, which offers representative information about the function's performance. Since the cloud computing market is heterogeneous and quickly changing, selecting the optimal provider is also not an easy task. In this context, heterogeneity means that service providers offer different service models with varying levels of resource monitoring and profiling support. This includes diverse granularities both in detail and in the format of the information offered by a provider.

This makes the situation unclear and unacceptable, and therefore we instead picture a different approach of solving this chicken or the egg dilemma. Out of the many open FaaS challenges summarised by Eyk et al., we are motivated to find out "how to replicate locally for testing the third-party, often closed-source cloud platforms"[vEIST17, p. 3]. We focus on finding more information about the resource demands of a cloud function prior to its deployment and therefore prior to the selection of a cloud provider. We think that independent information about the function is essential for the choice between the many combinations of possible providers and deployment settings. To this end, profiling the function locally and in isolation will reveal fine grained information about its resource demands. This information is ideally independent of both the machines used in the cloud infrastructure and the local experiment machine. It then serves as a decision aid for the cloud consumer.

1.2 Research Questions

This thesis researches the characteristics of a cloud function. The goal is to reproducibly collect the resource demands of any cloud function and model it independently of the hardware used in the experiment. Therefore, we are eager to answer the following research questions:

RQ1: Reproducibility How can we develop a reproducible profiling methodology for cloud functions?

RQ2: Model Resource Usage How can we measure and model the resource demands of a cloud function locally, but independent of local hardware?

RQ3: Provider Mapping What are the necessary steps towards mapping the provider resources as accurately as possible to a local experiment machine?

Essentially, the resulting model should provide insights about the cloud function under investigation with respect to its resource use. Regarding RQ1: Reproducibility, automating the approach as much as possible is important so that it can be used with only little overhead and is not prone to errors. We will investigate to what extent the model can be designed independently of the experimental hardware. Ideally, we will answer RQ2: Model Resource Usage by showing that the same experiment conducted on two machines will result in the same model. This model should be able to distinguish between CPU- and memory-bound cloud functions. RQ3: Provider Mapping is concerned with simulating the FaaS architecture used by the provider on a local machine. We will assess to what extent such a realistic mapping is possible and what abstractions need to be made in the process.

1.3 Related Work

As described in section 1.1, abstraction challenges still exist in the realisation of the FaaS paradigm. One challenge is to select the amount of provisioned resources, including memory, CPU, and network I/O, depending on the provider. As a solution, Jonas et al. suggest that either the developer receives more configuration options or the optimal settings of a function are automatically determined by the provider [JSSS+19]. The latter approach would contribute to the goal of the FaaS paradigm and ideally eliminate all operational tasks on the side of the consumer. For this, the authors name static code analysis, profiling previous function executions and dynamic compilation to different architectures as possible solutions. As with Microsoft Azure Functions automatic scaling of memory[2], this may hinder some developers who need more fine-grained control over the specific resources.

[2]Azure Functions scale and hosting documentation:
`https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/azure-functions/functions-scale.md`

Hellerstein et al. also identify and quantify critical drawbacks of FaaS architectures in its current form [HFG+18]. Most notably, stateless functions run in isolation and separate from the data they depend on. This necessitates a lot of I/O overhead in the form of network traffic to different services. Recent studies observed that the bandwidth on average is limited to about 538Mbps exemplary for Amazon Web Service (AWS) Lambda [KPL19, WLZ+18]. Other platforms like Google Cloud Platform (GCP) and Microsoft Azure showed similar results. The bandwidth can be further reduced by side effects like resource contention, where the assignment of multiple function calls (20 in this example) to a single Virtual Machine (VM) results in an almost linear slow down to 28.7Mbps [WLZ+18].

Substantial work has been carried out on troubleshooting and tracing functions. When creating a profile of a function execution, different levels of abstraction are distinguished. While some tools analyse the log output of an execution as well as additional information [PCZJ18], others take a more intrusive approach that directly examines the execution environment [CSL04]. In a higher level approach, Pi et al. utilised log messages and container performance data to find anomalies from the retrieved CPU, memory, disk I/O and network I/O resource data [PCZJ18]. By contrast, instrumentation tools provide specific and accurate information about the runtime performance of a system, but often depend on the language and environment used [CSL04]. Therefore, it is more difficult to reproduce the instrumentation approach in a general way. However, the reproducibility of an approach is a strong measure of its validity [FBS12, HHJ+12]. Consequently, an approach should be chosen that is highly reproducible and at the same time comes close to the accuracy of the instrumentation technique.

There are many threats to the reproducibility of systems research such as ad-hoc personal workflows, hard coded bash scripts or the limited availability of original code and dependencies [JSW+17]. As a solution, container technology helps to maintain a high level of reproducibility as it facilitates the automation of the experiment setup [Mer14]. By using containers, it is possible to revisit and replicate previous experiments easily, regardless of the hardware used [HHJ+12, Mer14]. At the same time, systems research is more objective, because many subjective evaluations of an experiment are possible [JMM+15]. Incidentally, container technology is especially suited for our approach as the target system uses containers to offer their FaaS services.

Kalibera and Jones observed shortcomings in the evaluation of systems research and proposed a method for improving experimental setups [KJ13]. They categorised influencing factors as variables that can be either controlled (chosen by the experimenter), random (changing with every iteration) or uncontrolled (imposed by the experiment environment). In addition to describing uncontrolled variables, they provide a statistically rigorous method to find the optimal number of repetitions for a benchmark that leads to robust results [KJ13]. For a correct categorisation, we must assess the sources of these differences and from there determine an appropriate sample size. For example, depending on the execution environment, I/O operations can either be categorised as random or uncontrolled variables. In the latter case, these variables must be identified and factored out as they could lead to a bias in the results [KJ13]. These undesired influences can be present at the container level, compilation level, execution level or within the context of a cloud function. Furthermore, it must be ensured that builds, executions and iterations are independent of each other.

Manner proposed a simulation framework that evaluates characteristics of a single cloud function [Man19]. It takes into consideration its load pattern as well as performance influencing factors implied by FaaS, such as the cold start overheads. The function, influential factors, and provider specifications are the inputs to the simulation. The result is a summary of the function properties and suggested settings for the execution environment i.e. the settings offered by a cloud provider. We will contribute to this simulation framework by investigating the CPU and memory I/O resource demands of a cloud function.

Previous approaches modelled the resource usage and other execution characteristics like function calls in numerous ways. The proposed techniques mainly differ in the type, granularity, and the scope of the observed resources. Ren et al. show that meaningful profiling is possible with only negligible overhead [RTM+10]. In their approach, they keep both computing and storage overhead to a minimum by aggregating the data in abstract models. Several approaches showed that being able to query obtained models is an important feature when detecting malfunctions, dependencies and interaction within resources. For example, Mace et al. modelled the function call graph using meaningful temporal relations like the "happens-before" property to understand dependencies better [MRF18]. Hauser and Wesner propose to use a Hidden Markov Model (HMM) to capture the relative resource use of a function in states which each describe a capacity utilisation [HW18]. The combined HMM represents the expected resource demands of a cloud function. It is then able to detect when resources are consumed disproportionally, for example, when there is a bug in the cloud function. While this approach facilitates alerting mechanisms, the model raises the abstraction level and is therefore incompatible with visual analysis tools like Grafana[3]. On an even higher abstraction layer, Leitner et al. model the deployment costs of cloud architectures which are composed of several cloud functions [LCS16]. In their approach named "CostHat", they use heuristics gathered from previous function executions to estimate the cost of one function execution. In the next step, they introduce a service call graph and calculate the total operating cost by accumulating the number of function calls and their relative costs.

Previous literature proposed various techniques to solving the numerous profiling and monitoring challenges present on cloud platforms. They see a lack of offline testing capability and [vEIST17, Eiv17]. We contribute to solving this challenge by investigating a non-intrusive offline approach approach to profiling. A model describing a cloud function is derived by applying some of the techniques discussed.

1.4 Structure

This thesis is organised as follows. In section 2, relevant concepts and used terminology will be described. Section 3 will look at profiling in general, paying particular attention to the differences between the provider- and consumer perspectives. Subsequently, the proposed local profiling approach will be outlined. It will be presented in greater detail in section 4. Based on an example experiment, the model will be described in section 5. Section 6 will evaluate of the results of the conducted experiments. Complementary to the evaluation, the approach will be discussed in section 7. Afterwards, a conclusion will summarise and assess the contribution and propose future work in section 8.

[3]Time series analytics tool Grafana: `https://grafana.com/`

2 Concepts and Terminology

Serverless is a cloud-based computing model which envisions the abstraction from hardware and operational concerns [vEIST17]. The event driven model Function as a Service (FaaS) adopts the idea of serverless applications and realises them through cloud functions. Eyk et al. define a cloud function as a "small, stateless, on-demand service with a single functional responsibility" [vEIST17, p. 2]. Cloud function instances are ephemeral, stateless, and only exist for the course of their execution, allowing them to automatically scale to a given demand. Apart from scaling, additional operational concerns such as managing resources and authorisation are the task of the cloud provider. They are responsible for controlling and optimising resource usage of the "measured service" by monitoring their infrastructure at suitable levels of abstraction [MG+11].

2.1 Monitoring and Profiling

Monitoring is the automated and long term technique used by operators of complex server infrastructures to get up to date information about its state. Their main goal is to ensure a high level of availability of applications and services. Monitoring is especially important in the serverless context as cloud consumers expect highly available services, while at the same time cloud providers face challenges like performance isolation, overhead optimisation and scaling [vEIST17].

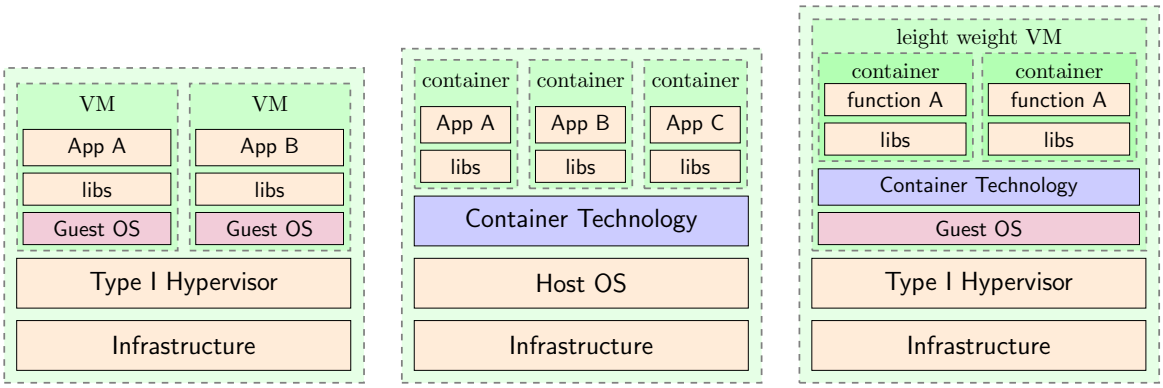
Profiling is more specific than monitoring: It is a dynamic code analysis technique that measures the use of computing resources at certain abstraction levels. In a low level approach, event-based profilers observe fine grained events like function-, instruction- and system calls, thread changes and object creation. This information must be provided by the used programming language or execution environment and may be represented as a call graph or as a flat profile. Both of these representations indicate the duration and characteristics of instruction or function calls. In contrast to these ideas, higher level approaches query different data sets of the execution environment at regular intervals. This data can be provided by the middleware, operating system, programming language, or execution environment resulting in a wide variety of granularity levels. With the addition of multiple abstraction layers, more overhead is contained in the profile which in the end make it less accurate [WBW15].

Also, resource contention in the form of a "noisy neighbour" can influence the profile in an undesired way. The profiling methodology itself can exert such an influence. This means that the overhead introduced by the profiling software influences the performance of the system under investigation. Profiling should therefore be carried out in isolation to increase the accuracy of the observed performance data.

With respect to cloud computing, recent literature favoured non-intrusive approaches as they cause less or no additional overhead on the application and are more versatile [ABDDP13, RTM+10]. The often unavoidable overhead must subsequently be quantified separately from the measurements and factored out accordingly [WBW15]. The next section will describe how container technology mitigates this issue by providing for resource isolation.

2.2 Container Technology

Linux Containers are a lightweight virtualisation technology that, unlike traditional VMs, do not require hypervisor abstraction. Since no hardware translation is used, the host operating system is shared with all containers as differentiated by figures 1a and 1b. Containers are isolated in terms of resources and their dependencies using technology features provided by the Linux kernel. Resource isolation means that the CPU, memory, and disc I/O resources can be explicitly assigned and monitored for each container. However, container technology can only provide this isolation to some degree, because resources are still shared with the host. Since the FaaS computing model requires a high level of granularity when allocating resources, the ability to limit container resources is critical. While resource isolation for network I/O is also possible, it entails further difficulties which will be discussed later. Apart from that, application isolation means that a container is self contained and does not have any external dependencies which are assumed on the host machine.



(a) Virtual Machine (b) Container Technology (c) FaaS Architecture

Figure 1: Architecture comparison between hypervisor based VMs, container technology and FaaS architecture.

The serverless computing model was made possible primarily by these two essential properties of container technology. Above all, container isolation makes the cloud functions independent of the hardware used, which facilitates the abstraction of the operational logic. Additionally, the stateless nature of containers natively enables the auto-scaling property of FaaS [vEIST17]. Based on the demand, new instances of a cloud function can be created or removed with very little overhead. External disk space used by container technology, for example in the form of Docker volumes, must instead be realised through third party services. The FaaS computing model isolates function instances inside containers using virtualised environments to enable its multi-tenant infrastructure. In concrete terms, providers execute cloud functions in containers running in "micro VMs" as depicted in figure 1c. These rely on purpose-built hypervisors designed to be as lightweight as possible. For example, AWS serverless offerings leverage the micro VM technology firecracker[4] to enable secure and isolated instances. They provide container- and function-based services in type-1 hypervisors [BLM19]. In doing so, they combine the security advantages of VMs with the performance advantages of container technology. Aside from that, this model also facilitates the monitoring and profiling of cloud functions.

[4]Firecracker GitHub repository:
`https://github.com/firecracker-microvm/firecracker`

Docker metrics	**definition**
stats_total_cpu_usage	sum of requested CPU times (nanoseconds)
cpu_i	individual CPU time (nanoseconds)
memory_usage	total current memory usage (bytes)
(r/s)x_bytes	bytes that are received / sent
(r/s)x_dropped	bytes that are dropped while receiving / sending
cgroups metrics	**definition**
user	amount of time a process has direct control of the CPU, executing process code (in ms).
system	amount of time the kernel is executing system calls on behalf of the process (in ms).

Table 1: Relevant metrics provided by both `docker stats` API and cgroups API. In our approach, we focus on the `docker stats` metrics.

In this work we use the popular container technology Docker, as it offers, besides easily accessible standard container functionality, a huge ecosystem of prebuilt containers [Mer14]. Docker relies on images which are built from a series of file system layers using the copy-on-write strategy[5]. An image is then initialised with a writeable file system layer to form a container. To achieve the desired isolation, Docker mainly uses the three Linux kernel features control groups (cgroups), namespaces, and change root (chroot). While cgroups limit and monitor the resource usage for a set of processes, chroot and Linux namespaces further isolate the container by changing the apparent root directory of a process and restricting the access to global resources. Two versions of cgroups are available. Since, as of now, Docker has not yet migrated to using cgroups version two, we decided to also use version one for our approach. As of now, Docker has not yet migrated to using cgroups version two, so we decided to also use it in our approach.

Docker offers resource limitation of a container in the following way. The user can specify the CPU resources as the relative CPU time assigned to a container. The CPU quota corresponds to the maximum share of virtual cores a container has access to defined by the parameter $cpus \in]0; |virtual\ core|]$. For example, for a machine with four virtual cores, the user can limit the CPU time to $0\% < cpus \leq 400\%$. Regardless of the setting, the CPU time is measured and assigned across all available cores. To limit a container to only access a specific set of cores, Docker offers to set the CPU affinity. This allows the user to pin a container to one or more virtual cores. Both of the CPU resource limitations work additively. By analogy, the memory resources allocated to a container can be limited according to the user's needs. Docker offers both a hard and soft limit for fine grained memory allocation. The former is the maximum amount of memory available, whereas the latter comes into effect when the host is running low on memory. This also shows that container technology does not natively isolate the resources. Instead, it is able to limit the total resource use of a container. Full resource isolation entails abstracting from the tasks running on both the host and in other containers. This implies that resources are assigned exclusively to a container.

[5]About storage drivers: `https://docs.docker.com/storage/storagedriver/`

In addition to the core functionalities, Docker offers profiling capabilities by means of the `docker stats` API[6]. While many more metrics are provided by this API, the most relevant ones for us are the CPU and memory utilisation of a container. As for the CPU metrics, our profiling approach also considers complementary information provided by the cgroups API. Compared to Docker representing the requested CPU time, the CPU metric offered by cgroups represent the total amount of CPU time consumed by a control group [FFRR15]. We observed that the measured cgroups time is therefore marginally higher than the CPU time provided by Docker. While our profiling prototype presented in section 4 supports both APIs, we will only consider the `docker stats` API in the assessment. Thereby, we will not introduce time stamp issues that could affect data consistency.

[6]Docker Stats reference:
`https://docs.docker.com/engine/reference/commandline/stats/`

3 Profiling

With the growing complexity of server infrastructures, profiling and monitoring is becoming an increasingly important part of understanding the interlinked computing resources in both existing and new cloud computing models. This ability of gathering data about the underlying system behaviour is referred to as operational visibility [OSN+17]. It is crucial for both controlling and managing the hardware and software, but also to comply with Quality of Service (QoS) requirements defined in a service level agreement [ABDDP13]. Without operational visibility, both cloud providers and consumers have no way to judge the health and behaviour of a cloud infrastructure or a specific service. However, profiling server infrastructure always implies an additional overhead which might even influence the performance of the offered services [RTM+10]. Nevertheless, meaningful information about the state of the cloud infrastructure is crucial, so this overhead is more than necessary.

This section looks at both the cloud provider and the cloud consumer perspective and compares the available profiling techniques. Furthermore, it will describe and assess the proposed local profiling approach and address its limitations.

3.1 Abstraction Layers

Cloud providers monitor their infrastructure on multiple levels to asses the resource usage of all types of compute instances. These include all existing service models, which are becoming increasingly numerous and complex [FGJ+09], but ideally are also flexible to additions and changes. While several approaches apply to multiple models, we will primarily focus on the FaaS computing model in this work.

As mentioned in section 2.1, a distinction can be made between low level and high level monitoring approaches [ABDDP13]. On a low level, providers monitor the health and the performance of a physical system to detect malfunctions and resource interferences as soon as possible as well as plan for unpredictable loads. Moreover, high level monitoring is used to collect information about the virtual execution environment, for example a cloud function instance running on a VM. Even higher level approaches leverage dynamic and static instrumentation to extract fine-grained information about an application. Recent literature proposed a large number of approaches, most of which are non-intrusive. An overview of the techniques using several abstraction layers can be seen in table 2.

On a high level, Mace et al. combine dynamic instrumentation with available hardware tracing techniques [MRF18]. They see a lack of available profiling techniques that do not reflect the expectations and incentives of developers. Also, they name the low data granularity as a problem. To address these two issues, they propose to dynamically instrument the call graph of an application and thereby enable queries. Oliveira et al. present the profiling approach designed and used by IBM in the Container as a Service (CaaS) computing model [OSN+17]. They fetch performance data and container images periodically, which results in a considerable overhead. In doing so, they mainly focus on addressing security concerns, which are crucial in the multi-tenancy nature of CaaS [HFG+18]. A vulnerability in one container can impair the performance and integrity of services running in adjacent containers on the same host. For example, researchers name the “fork bomb”

Abstraction Layer		Approach
intrusive	instrumentation in programming language (e.g. Java)	Dynamically or statically instrument events inside an application. [MRF18, MKW19]
non-intrusive	virtual layer (e.g. docker container)	Periodically inspect the state of virtual resources by using APIs specific. [PCZJ18, CP17, OSN+17]
	physical layer (e.g. operating system, hypervisor)	Periodically inspect the state of system using operating system tools. [RTM+10, CP17]

Table 2: Comparison between monitoring and profiling approaches using different abstraction layers.

as a containerized security threat, where a malicious container recursively forks new processes. Because Linux based operating systems limit the number of processes to 2^{15} for 32-bit and 2^{22} for 64-bit architectures, undefined behaviour arises when exceeded. In their distributed troubleshooting approach, Pi et al. also focus on the container abstraction layer [PCZJ18]. However, they put more emphasis on meaningful logs instead of just runtime and container metrics. By combining both the fine-grained container resource metrics and log messages in so called keyed messages, they are able to enhance the quality of the logs. The retrieved data can be queried a posteriori to reconstruct the workflow and find causes for anomalies. Casalicchio and Perciballi lower the abstraction layer even further by using both the virtual and the physical execution layer [CP17]. They provide a comprehensive overview of techniques available for profiling and monitoring docker containers. Both high-level metrics provided by `docker stats` and `cAdvisor`[7] and low-level metrics provided by the operating system were used. Ren et al. presents the "Google Wide Profiling" technique which is deployed on all of their systems [RTM+10]. They use OProfile[8] to gather several low level events like CPU cycles, cache misses, branch mispredictions, or heap memory allocations, only to name a few. Combining metrics in several models helps them to find frequently used shared code and improve hardware utilisation. They strongly discourage the use of higher level intrusive approaches, as service consumers expect high QoS without unnecessary overhead. Because of the heterogeneity of offered services, a higher level approach implies that it is tailored to specific computing models. Therefore, it may only give insights about a subset of services.

The literature review suggests that a non-invasive approach is appropriate when general information about the resource demands of a system is to be obtained. This is particularly relevant when we do not have any information about the software. Higher level approaches using instrumentation techniques may be more specific and accurate, but are less versatile and not suitable for our containerized approach.

[7] cAdvisor – Analyse resource usage and performance characteristics of running containers: `https://github.com/google/cadvisor/`

[8] OProfile – A System Profiler for Linux: `https://oprofile.sourceforge.io/`

3.2 Cloud Provider Perspective

In the FaaS context, profiling entails measuring the resource usage of an isolated cloud function instance running inside a container. It mainly consists of the CPU, memory, network I/O resource consumption as well as static information, for example the execution time of the function. In the other service models that additionally make use of permanent storage, disk I/O is also monitored. The presence of a "measured service", where the consumer pays only for the resources used and not for the resources allocated, are two essential characteristics of the FaaS paradigm [MG+11, JSSS+19]. To this end, FaaS billing models combine both the CPU and memory resource usage with the execution time[9].

Only high-level information, which varies in scope and granularity from provider to provider, is made available to the cloud consumer via an API [ABDDP13]. This limited availability of monitoring information restricts the trust relationship and leaves some questions unanswered. With regard to the multi-tenant nature of FaaS, there exist security concerns as well as problems like the aforementioned "noisy neighbour" effect [vEIST17, HFG+18]. While former concerns were addressed in section 2.2, the latter concerns are more relevant in this context: Since compute resources are not allocated exclusively, the performance of a function may degrade if a collocated function consumes resources disproportionately. Additionally, overbooking is an essential technique for FaaS providers to increase resource utilisation. This results in a high deployment density, where providers strike a balance between efficient resource usage and useful resource guarantees like scalability [USR09]. Although all the information cloud providers collect would be important to its consumers, it is not offered primarily for security reasons. In general, they only provide an isolated view of the resources. As the cloud providers have control over the information offered, the consumer must trust the limited information made available to them.

3.3 Cloud Consumer Perspective

In a strict realisation of the serverless paradigm, the cloud provider will assume all tasks needed to provide the operational logic like managing resources, scaling, and authorisation. Under this assumption, the provider is fully transparent and always finds the optimal setting with respect to resource allocation, load balancing, and scaling. While the provider mainly focuses on optimal resource allocation, ideal for the consumer means that the desired service is highly available with the minimal use of paid resources. However, developers must still quantify the resources requirements of a function and specify the deployment region [ABDDP13, JSSS+19]. This difficult decision is very important as it has a significant impact on performance and costs. On the one hand, over-provisioning leads to resources not being utilised to their capabilities resulting in higher costs. On the other hand, provisioning too little resources has a considerable effect on performance and availability, which can lead to unpredictable states. For example, a function execution can be terminated because the maximum execution time has been exceeded or processes are killed due to too little memory.

[9] AWS Lambda Billing Model: `https://aws.amazon.com/lambda/pricing/`
Azure Cloud Functions Billing Model:
`https://azure.microsoft.com/en-gb/pricing/details/functions/`

Jonas et al. characterise resource abstraction challenges that are inherent in the serverless paradigm [JSSS+19]. In particular, they see this lack of abstraction as a problem, where the consumer is faced with the difficult task of choosing an appropriate resource setting that in the end can be very impactful on the bill. The proposed approach to profiling previous executions is based on the assumption that the cloud function is deployed beforehand and there exist representative executions that can be analysed. Strictly speaking, this again requires the definition of initial resource requirements.

The availability of profiling tools can lead to a tight coupling between cloud providers and consumers. To solve the problems introduced by this heterogeneity, cloud monitoring adapters can enable a more provider-agnostic technology. As with other interoperability challenges, there is currently no feasible monitoring adapter approach for the FaaS model [vEIST17]. The cloud consumer thus depends on the limited profiling tools offered by cloud service providers.

3.4 Profiling Cloud Functions Locally

In order to avoid the unpredictability resulting from possible misconfiguration of a cloud function, more information about its characteristics is needed. For a more objective decision making process, the precise resource usage of a function must therefore be known prior to deployment. Out of these, the CPU-, memory-, and network I/O resource demands are most relevant to FaaS.

To this end, the local execution of a function with arbitrary load can give relevant insights about its resource demands. We propose to profile cloud functions in a local, non-intrusive approach by utilising the lightweight virtualised environment offered by container technology. The goal is to simulate the production environment in isolation and describe the resource characteristics independently of the hardware used in the experiment. The function under investigation is executed a number of times with a sample request and well defined available resources. By means of an a priori calibration step, the provided resources can either correspond to a specific cloud provider setting or be defined manually by the consumer. During its execution, the resource demands are recorded via APIs provided by the used virtualisation technology and the operating system: In our approach, the measurements contain the relative CPU usage and the memory usage. While we also record the network traffic, it will not be represented in our model. The accumulated metrics are saved as a time series database in an accessible format like CSV. Additionally, static meta information such as start and end times and whether or not the function was executed normally are obtained and stored along side the profile. For example, a lack of memory could lead to an unexpected termination of the container by means of the Out of Memory (OOM) manager of the host[10]. Auxiliary information such as logs to standard output or status codes can also be retrieved.

This approach combines several important objectives. First and foremost, the simulation comes close to the production environment: Just like cloud providers, it also runs the cloud function inside a container. Assuming that the experiment machine is more powerful than the container running in the cloud, we are able to restrict the local con-

[10]Exemplary for Docker, the kernel by default kills processes inside a container if an OOM error occurs. Docker run reference describing the behaviour of the Docker container based on different memory settings: `https://docs.docker.com/engine/reference/run/#user-memory-constraints`

tainer resources to an arbitrary setting offered by the provider. In addition, container technology enables the reproducibility of the experiment independent of the hardware. By independently building and running containers, random influences are eliminated as much as possible [KJ13]. Jimenez et al. describe a research experiment as a triplet of (1) workload, (2) specific system the workload runs on and (3) results from a particular execution [JMM+15]. The reproducibility of measurements in systems research therefore heavily depends on the availability of both the workload and the system under investigation (or simulation of the systems). They propose to solve this issue by mapping the original hardware to another system. By utilising a self contained experiment setup in combination with the calibration step, our approach is able to abstract from these challenges using this mapping. Finally, another advantage of our approach is its independence to cloud providers. Several authors suggest to analyse cloud functions during and after its deployment [JSSS+19, ABDDP13]. While the realistic environment allows fore a more realistic assessment, we still see drawbacks compared to a local approach. The main disadvantage is the heterogeneous and limited operational visibility. Also, these approaches in most cases only consider a subset of available FaaS providers. With the mapping of the cloud environment to a local machine, we facilitate a more exhaustive analysis, since the profiling step is conducted independently.

Our prototype which implements this profiling approach is described in section 4.

3.5 Limitations

The performance of a containerized application is influenced by several factors. Therefore, it is not possible to profile a cloud function in perfect isolation. For example, in addition to the function under investigation, a container or its host might run other tasks. As these can negatively influence the shared resources of the container, the host can be the "noisy neighbour" of a container.

With the hardware independent calibration step, the local environment at first glance has the same amount of resources as the container running in the cloud. However, if the cloud function makes use of algorithms that are optimised on some hardware, we are unable to predict its performance in the cloud solely from the calibration step. For a mapping to be possible, both the calibration step and the cloud function itself need to be hardware independent. We will assess the hardware independence of a cloud function as well as classify its impact on the validity of the model in section 6.

While the resources CPU and memory can be controlled and measured precisely, there still remain many differences between the provider and a local containerized environment. As of now, it is for example not possible to limit the network I/O bandwidth of a Docker container natively[11]. The latency to adjacent services can also not be controlled without significant overhead. Possible approaches that isolate network I/O traffic will be outlined in section 8.3.

[11] Discussions to add runtime option to throttle network I/O bandwidth:
`https://github.com/moby/moby/issues/37`
`https://github.com/moby/moby/issues/4763`

4 Methodology

We developed a prototype that profiles a cloud function in a repeatable way. In order to ensure its reproducibility, we define an experiment and all required parameters in a metadata file using the machine readable format JSON. An example experiment can be seen in listing 4, which we will also use in the following. It contains all the information necessary for the experiment, which is divided into a calibration step and a profiling step. The first step allows for a comparison between the provider and the local environment and requires the definition of allocated resources used for the calibration step. For the second step, the parameters of the examined cloud function, i.e. a representative function call, need to be defined. Latter configuration is therefore specific to our load simulation function and must be adjusted when other cloud functions are being profiled.

To facilitate a comparison and validation of our approach, the experiments were conducted on two machines, which we call primary and secondary machine as summarised in table 4. Unless otherwise specified, we will refer to the primary machine. The main method is shown in listing 1, where an experiment consisting of the calibration and the profiling step is carried out.

```
1 Experiment experiment = Experiment.fromFile("experiment.json");
2 experiment.calibrate();
3 experiment.profile();
```

Listing 1: Main method of the containerized profiling approach consisting of the two steps calibration and profiling. The experiment definition can be found in listing 4.

In the following, we will describe the load simulation approach used to test the containerized profiling tool. We will then discuss the calibration step which is used in combination with the obtained metrics to achieve a hardware independent model. Figure 2 provides an overview of the prototype, which is available on GitHub[12].

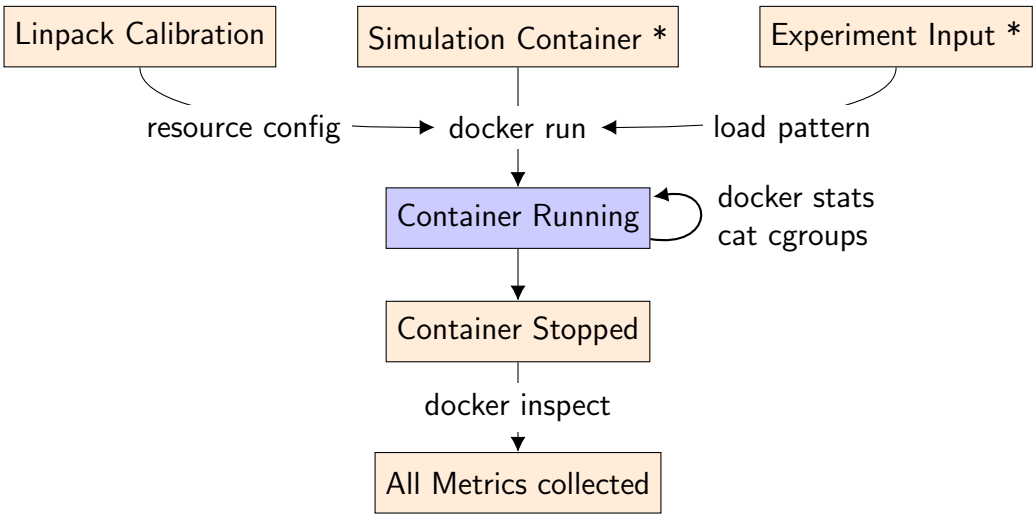


Figure 2: Overview of the metrics collection process. Steps marked with * are specific to our experiment.

[12] Profiling Cloud Functions:
https://github.com/martin-endress/profiling-cloud-functions

4.1 Cloud Function Simulation

We developed a Docker container which runs a load simulating cloud function. Based on the experiment input, which represents one function call, the container is able to simulate CPU-, memory-, and network I/O resource consumption in a controlled manner.

In general, we assessed two approaches of defining the cloud function image. The Docker image can be defined manually by specifying the command executing the cloud function in a `Dockerfile`. Based on this approach, the `Dockerfile` of our simulation image is defined in listing 3. Alternatively, an automatic approach can utilise the Java class loader to abstract from this manual step. This way, the cloud function residing together with required libraries in a jar archive is loaded and executed automatically. Although the second approach comes closer to the provider setting, where the consumer only needs to provide the function archive and the function name, we observed that it imposes a disproportionate overhead. This hinders a comparison among multiple machines and the FaaS environment, which exhibits a lower overhead. From now on, we will therefore only look at the manual approach, which executes the cloud function directly.

In the prototype, we chose Java as the programming language for the example cloud function simulating resource demands. The utilisation of all resource types is simulated individually and executed in parallel in the container. The implementation of each single-threaded load simulator is trivial and briefly described in the following:

CPU load (relative) The simulation function executes a load pattern which defines the percentage of CPU usage over a given time period. Apart from the varying overhead caused by the Java Virtual Machine (JVM), this simulation always produces the same results regardless of the machine used.

CPU load (absolute) This load simulation recursively calculates the n^{th} value of the Fibonacci sequence. Contrary to the relative CPU load, the performance depends on the CPU resources present.

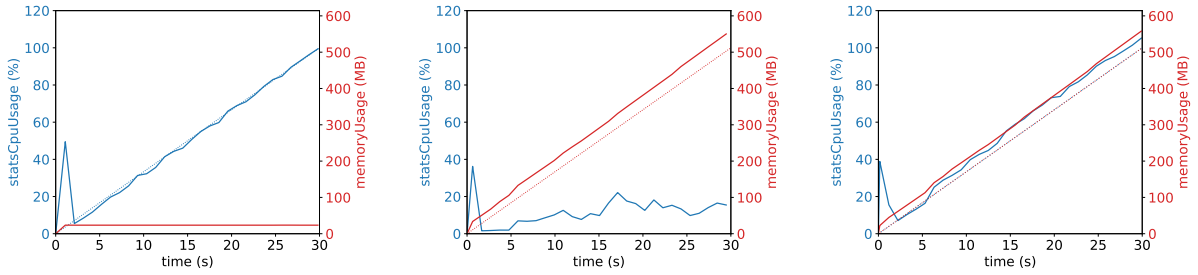
Memory load The amount of memory defined in the load pattern is sequentially allocated. Due to the unpredictable nature of garbage collection in Java, memory is only allocated and never released in this simulation.

Network I/O load The network I/O load pattern includes the amount and size of packages that are to be transmitted over the network. In addition to the load pattern, the target endpoint as well as query parameters need to be specified.

To simulate network I/O load in a more controlled manner, we implemented a containerized service mock similar to Mocky[13]. It generates HTTP responses that have a certain size and can be delayed depending on the request.

We did not include disk I/O simulation, as the FaaS paradigm explicitly refrains from permanent storage. Besides, there is only little support and several open challenges with respect to profiling disk I/O usage using containers [CP17].

[13] Mock your HTTP responses to test your REST API:
`https://github.com/julien-lafont/Mocky`



(a) CPU load (relative) (b) Memory load (512MB) (c) Combined load

Figure 3: Exemplary executions for CPU and memory loads. The dotted line represent the expected resources usage.

Exemplary load simulations for each of the resource types can be seen in figure 3 and in figure 11. Each of the functions displayed runs for 30 seconds and simulates a load on one type of resource defined in the load pattern. The measurements are obtained from the `docker stats` API, and should correspond to the load pattern drawn as a dashed line. As shown in the figures, there is an additional overhead in the other resources apart from the defined load pattern. While a small deviation from the expected resource consumption can be seen in all experiments, it is most evident in the combined simulation shown in figure 3c. This overhead shows some of the drawbacks that we discovered in our Java based simulation approach. In addition to discussing these shortcomings, we propose improvements to this simulation approach in section 8.3.

4.2 Calibration Step

The CPU resource metrics are expressed as the proportional CPU time of a container or process. This relative metric is therefore only applicable to the hardware used in the experiment and cannot be compared among different architectures. In order to nevertheless quantify the CPU resource demands of a function independently, we introduce a calibration step prior to the profiling of the function. Back and Andrikopoulos used a similar approach to calibrate their system [BA18]. They used this approach to compare different cloud providers more objectively with the help of a microbenchmark. We build on this approach by incorporating the performance of local machines into the comparison.

We have decided to use Linpack as our calibration method, since it is a popular and often used benchmark for measuring the processing power of a machine [DMBS79]. For example, it is used for the TOP500 list which quantifies the performance of the 500 most powerful supercomputers[14]. Linpack solves a dense linear system in single or double precision arithmetic and represents the measured computing performance as the number of Floating Point Operations per Second (FLOPS). Both the implementation and the resulting metric are hardware independent and can therefore be used to compare multiple physical or virtual machines. Hardware independence means that the algorithm uses the CPU resources in isolation and does not benefit from optimisations that are available on certain hardware. The unit of the measured computing power is also hardware independent as it represents the absolute number of floating point operations that can be performed on the machine.

[14]TOP500 list comparing the performance of supercomputers: `https://www.top500.org/`

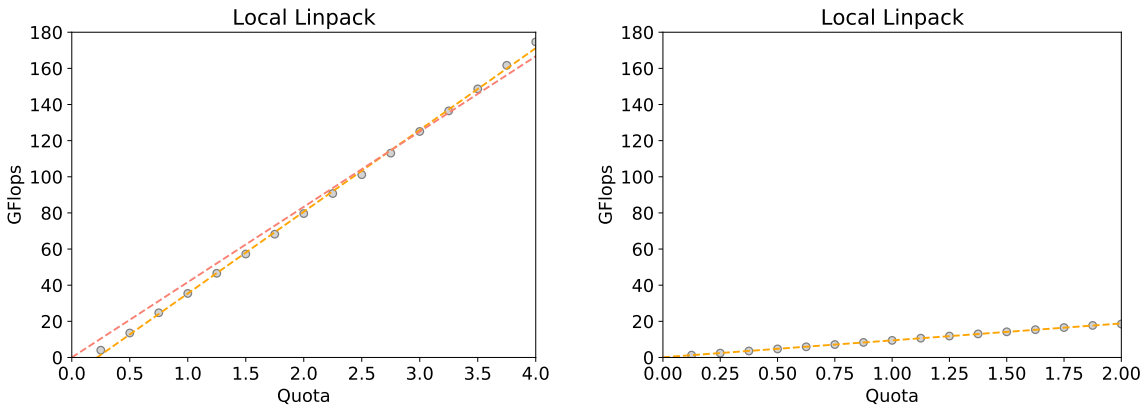


Figure 4: Local Linpack benchmarks on primary(left) and secondary(right) machines with limited CPU quotas. Orange and red lines represent correlations (see table 3).

In an effort to achieve a high level of resource isolation, we aim to profile the cloud function using specific and fine-grained resource settings. To accomplish this, we run the local calibration in a Docker container using various resource limitations and observe the measured performance of each setting. On both the primary and the secondary machine, we executed the benchmark with a wide range of available CPU quota limitations. Because Linpack is multi-threaded but not hyper-threaded, it runs one thread per physical core present on the experiment machine. Since the primary machine has four physical cores while the secondary machine has only two, we decided to run the benchmark with CPU quotas ranging from 25% to 400% and 12.5% to 200%, respectively. Also, as we did not expect the performance of a machine to change over time, we executed only one Linpack benchmark per resource setting. The results of the local calibration step can be seen in figure 4, where each of the dots represents one Linpack measurements based on a CPU quota limitation.

We expected the measured performance to be proportional to the assigned CPU quota of experiment runs. After this assumption was confirmed as shown in figure 4, we quantify it by constructing a linear regression model and calculating its correlation. Consequently, the yellow lines represent the correlation between the CPU quota and the measured performance in GFLOPS. The resulting correlations for both machines are summarised in table 3. As the proportionality of available resources with CPU performance is an intrinsic assumption of our calibration approach, we also considered removing the intercept from the linear regression models. For comparison, the resulting correlation for the primary machine is plotted as a red line in figure 4. We regard this approach as feasible, because we know that the origin is part of the model. Theoretically, assigning 0% container quota will results in no performance. We will from now use the models with the intercepts and evaluate this idea in section 6.4.

Apart from the local Linpack measurement, we deployed and executed the benchmark in the cloud, exemplary on AWS Lambda. This allows us to directly compare the processing power of a local machine with the machine used by the provider. Since the same experiment setup has also been used by Malawski et al., we are able to compare both results with each other [MFGZ17].

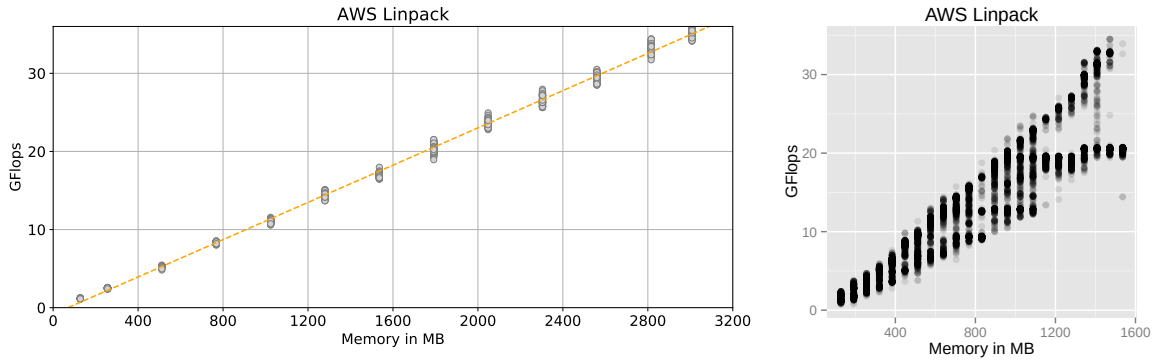


Figure 5: Repeated execution (left) of the AWS Linpack benchmark compared to the original results from 2017 (right, [MFGZ17]).

Calibration		R^2	intercept	slope	Interpretation
local	Primary Machine (see figure 4 left)	99.92%	−9.9	45.27	GFLOPS per additional CPU core
		99.10%	0*	41.67	
	Secondary Machine (see figure 4 right)	99.97%	0.10	9.36	
		99.96%	0*	9.43	
AWS	AWS Lambda (see figure 5 left)	99.77%	-0.83	0.0119	GFLOPS per additional MB of memory
		99.62%	0*	0.0115	

Table 3: Correlation between the CPU quota or memory size and the CPU performance for both local and AWS Lambda deployments. Models with intercepts marked with a * do not include an intercept.

On AWS Lambda, the computing resources cannot be directly adjusted and scale linearly with the selected memory size. We therefore deployed 13 Linpack cloud functions with memory sizes ranging from 128MB to 3008MB in increments of 256MB. The resulting memory settings provide a broad spectrum of the available granularities of resource configurations including the minimum and maximum setting[15]. Since the timeout of the API gateway is limited to 29 seconds[16], we temporarily stored the results of the benchmark in an AWS S3 bucket and retrieved them from there. Contrary to the local benchmark, we expect the performance to vary because of influential factors like performance degradations and differences in hardware used. Hence, we decided to perform the benchmark 75 times for each memory setting over. Compared to the 12,000 benchmark executions in the initial experiment, the resulting 975 benchmark executions still render a small sample size. Apart from the benchmark results, we also fetched information about the used VM. This allows us to classify the processor and gain insights into the provider's VM allocation policies. To this end, we used the suggested VM identification technique using the start time of the VM as a heuristic for its identity [LRC+18]. It is stored in the `/procfs` file system and, in addition to identifying the VM instance, it reveals information about the CPU used. However, the results may include false positive identifications, when two distinct VMs are started at the exact same time. For now, we assume the feasibility of the approach, since the authors have calculated a very low probability for this event [LRC+18].

[15] AWS Lambda Limits:`https://docs.aws.amazon.com/lambda/latest/dg/limits.html`
[16] AWS Gateway Limits:
`https://docs.aws.amazon.com/apigateway/latest/developerguide/limits.html`

The results of both the initially conducted benchmark by Malawski et al. and our rerun can be seen in figure 5. Again, each dot represents one performance measurement based on one memory setting. Based on the results of the provider calibration step, we again calculate a linear regression model to find the correlation between available CPU resources and computing performance. We calculate the correlation between the selected memory size and the CPU performance measured by the Linpack benchmark. Using the performance models of both local and provider environments in table 3, a direct comparison is now possible. We will use these obtained performance models in section 5 to map the provider performance to the local containerized environment.

4.3 Collecting Resource Usage

After the definition of the cloud function simulation image and the calibration step, a container is created and started with a specific experiment input. The input consists of the load patterns for each resource type as well as resource limitations of both CPU and memory. It is passed as parameters to the Docker run command, after which the cloud function is running in the container.

To then collect its resource demands, our prototype utilises two information sources: Primarily, the `docker stats` API is queried. Additionally, it collects information offered by the Linux kernel technology cgroups resulting in the combination of collected resources as summarised in table 1. We queried the `docker stats` API which offers the current metrics per container every second. After receiving the response, we fetched the cgroups information and merged the two together. Assuming that both information sources are up to date when receiving the metrics, no distinction was made between the two timestamps. We observed a maximum delay of three milliseconds between querying the two APIs. When the container has terminated, the metrics collection stops and the complete time series database is stored as a Comma Separated Values (CSV) file. Each line in the file represents the resource consumption in one second, which is the maximum temporal resolution of the `docker stats` API.

After termination, additional meta information about the function execution is collected. We queried the `docker inspect` API offering us general information about the state of a container. We store the exit code, the duration of the function execution, and information about its available resources. Additionally, we calculated the total as well as the average amount of resources consumed relatively to the resources assigned to a container.

5 Model

As previously discussed, the resource demands of a cloud function are obtained relative to the host's hardware. Because the CPU utilisation is measured as a percentage of CPU time, it can only be considered with respect to the processor used. Consequently, we cannot compare the metrics offered by the operating system or container technology among different experiment machines. While the available metrics for absolute memory and network I/O demands are independent, their speed and bandwidth again depend on the hardware used.

We therefore propose a model which abstracts from as many of these dependencies as possible.

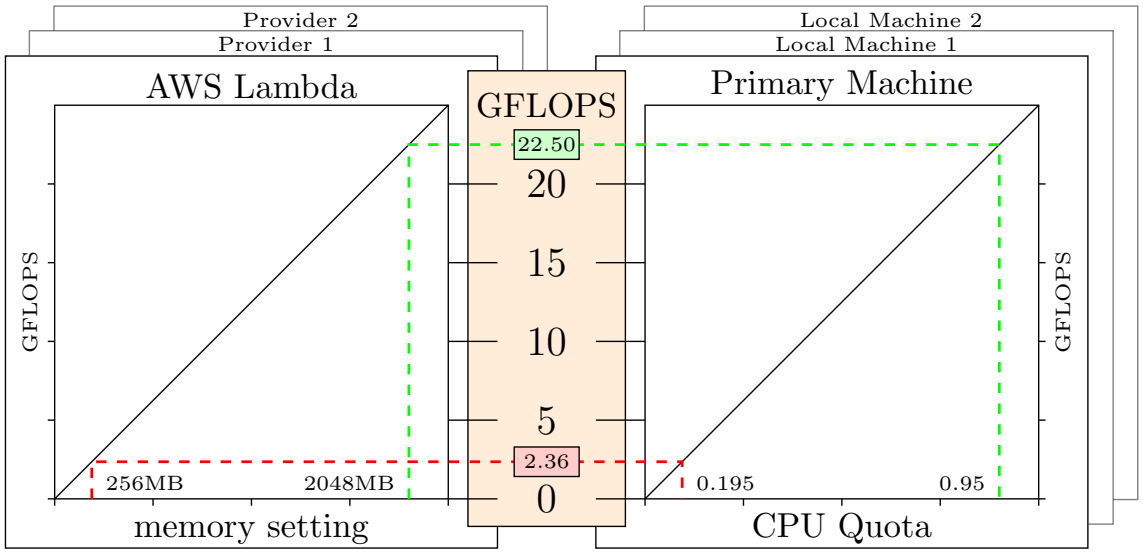


Figure 6: Mapping from cloud provider performance to local container quota via the independent metric GFLOPS. The example shows the mapping from AWS to a specific experiment machine (table 4).

5.1 Provider Mapping

The first step of the model derivation process is shown in figure 6, where we map the computing resources measured on the cloud platform to a local experiment machine. This mapping from the cloud provider environment to the local machine uses the respective resource models derived in the calibration step. On the provider side, the computing performance is dependent on the memory setting. Whereas on the local machine, we control the computing performance by means of container limitation. The calculated correlations in the resource models are specific to a provider offering or the local machine used, indicated by the interchangeable layers in the figure. Especially on the provider side, they may change over time and must therefore be calculated in regular intervals. Since both models describe the correlation with the hardware-independent metric FLOPS, a specific memory setting on the provider side can be associated with a container CPU quota on the local machine. Obviously, this requires the local machine to have more resources to begin with.

The set of simulated provider memory settings has to be defined in the experiment metadata, for example listing 4. Together with the performance models obtained from the calibration step, it is the input to the model mapping. While these memory settings may match provider offerings, we choose to simulate the 256MB and 2048MB settings offered by AWS Lambda. Therefore, we use the AWS Lambda resource model from table 3 to estimate the performance of a memory setting in GFLOPS. The mapping of the two settings is represented as red (256MB) and green (2048MB) lines in figure 6.

By using equation 1, we can estimate the container quota corresponding to a memory setting of a cloud function on AWS Lambda. In the formula, $f_{environment}$ refers to a performance model in table 3. Based on the available resources memory or container quota, it estimates the performance in GFLOPS. By analogy, the inverse function f^{-1} estimates the resource setting corresponding to the estimated performance.

$$container\ quota = f_{local}^{-1}\left(f_{provider}\left(memory\right)\right) \tag{1}$$

The model estimates that the 256MB and 2048MB Lambda instances are assigned computing resources of 2.36GFLOPS and 22.50 GFLOPS, respectively. Based on this hardware independent metric and a local performance model, the corresponding CPU quota is calculated for a specific experiment machine. While we calculate the CPU quotas for both the primary and secondary experiment machine, figure 6 only shows the local performance model for the primary machine. As a result, the 256MB setting corresponds to a local quota of 19.5%, whereas the 2048MB setting corresponds to 95.0% on the primary machine.

```
1  public void profile() throws ProfilingException {
2     statsRetriever = new StatsRetriever(experimentName);
3     for (int memory : simulatedMemory) {
4        double gFlops = providerPerformanceModel.getGflops(memory);
5        double quota = localPerformanceModel.estimateQuota(gFlops);
6        if (quota < 1){
7           throw new ProfilingException("Not enough resources.");
8        }
9        ResourceLimits limits = new ResourceLimits(quota, memory);
10       statsRetriever.profile(loadPattern,limits, numberOfLoads);
11    }
12 }
```

Listing 2: The Method `profile` creates a machine specific execution model and, if possible, profiles the cloud function with corresponding resource limits.

In summary, we obtained the machine specific execution model for a given memory setting consisting of the assigned memory and CPU quota. The implementation is shown in listing 2, where the first five lines are responsible for this provider to local machine mapping. After creating the profiling instance called `StatsRetriever`, we iterate over the specified memory sizes and perform each resource mapping. Based on the `memory` size, the `providerPerformanceModel` calculates the independent metric `gFlops`, which is then used to estimate the corresponding container `quota` on a local machine.

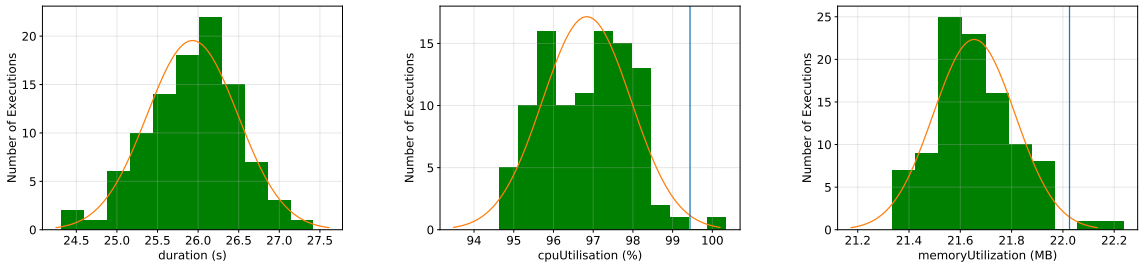
The potential use of multi-core VMs at the cloud provider introduces additional complexity in the mapping process: For example, function instances on AWS Lambda v2 have access to two virtual cores. Analogous to container CPU quota limitation, we suspect that AWS imposes a total limit in execution time across several virtual cores. Based on this assumption, we propose our selected approach and highlight its advantages and disadvantages. In our prototype, we determined that the simulation of a cloud function is only feasible if the estimated CPU quota is less than one virtual core (100%). If this is the case, the container running the cloud function is assigned the calculated quota. Otherwise, we decide that the mapping is not possible. This abstracts from potentially invalid assumptions about the function. We expect that both single- and multi-threaded functions will be realistically mapped in this way. The locally obtained model will examine similar performance regardless of the concurrency characteristics of a cloud function. For example, a task using two threads each consuming 25% of the CPU will result in the same model compared to a single-threaded task accessing 50% of the CPU. However, it might be the case that a single-threaded function will not utilise the potential computing resources of a multi-threaded VM.

A realistic mapping of the execution environment mandates a more fine-grained limitation of virtual cores. This would require changes in the calibration step, as Linpack only uses one thread per physical core. Further difficulties arise when the resources of individual cores are to be assessed and simulated locally. It is limited by the resource isolation provided by container technology, which currently does not offer fine-grained CPU pinning.

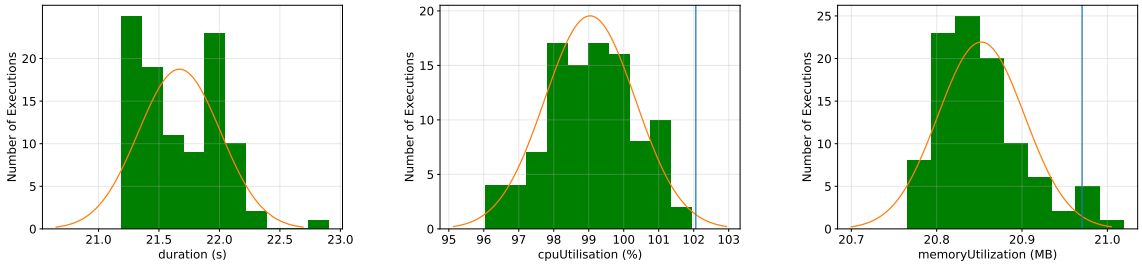
5.2 CPU and Memory Model

In the next step, the previously obtained local execution parameters are used to run a cloud function and generate a profile in the context of this model. For this, the function is run inside a container with said resource limitations using a representative load pattern. In our experiment, this load pattern is the input to the simulation and also part of the metadata as shown in listing 4. Subsequently, its runtime performance is measured and stored as described in section 4.3. Although these absolute resource usage metrics are still hardware dependent, they can now be associated with the independent model. Therefore, we can compare the relative resource demands of CPU and memory I/O independently of the experiment machine.

To demonstrate this comparison, we conducted two experiments on the two machines, each simulating two memory settings. The CPU intensive experiment runs the absolute CPU workload and calculates the 45^{th} number in the Fibonacci sequence. In addition, the mixed workload allocates 64MB of memory as defined in listing 4. Figures 7 and 12 show the results of the two experiments, in which the experiment functions are executed 100 times. They plot the execution durations, CPU utilisation and memory utilisation of each run as histograms. In perfect isolation and assuming determinism, we expect the same output in each run on both of the machines. However, this is not the case as many different influences lead to a normal distribution of the observed results within one machine, drawn as an orange line. When comparing the two experiments with each other, further differences can be noticed. The difference in execution time of 19%, which we primarily explain by the limited resource isolation, will be analysed and assessed in the next section.



(a) Primary machine execution duration, CPU utilisation and maximum memory use model



(b) Secondary machine execution duration, CPU utilisation and memory use model

Figure 7: Model resulting from the mapping experiment running the CPU intensive load simulation comparing primary and secondary machine (n=100). The orange line represents the normal distribution, whereas the blue line indicates the 99% quantile for CPU and memory models.

Using these models, we are able to make several observations about the CPU and memory demands with respect to the performance model: The model shows on which type of resource the cloud function depends. In essence, it indicates if its performance is bounded by CPU or memory. To obtain this information, we calculate the relative resource use at a certain quantile of the normal distribution. In our experiment, we choose and plot the 99% quantile. In concrete terms, the model concludes that the observed CPU intensive cloud function is 99.5% dependent on CPU resources and 8.6% dependent on the memory. The memory dependence gives limited insights, as it only considers the total amount of memory used. Analogous to CPU utilisation, a metric reflecting the total time spent accessing memory would instead be more relevant.

6 Evaluation

We conducted two experiments on the primary and secondary machines. The CPU intensive task calculated the Fibonacci sequence, whereas the mixed task additionally allocated a fixed amount of memory over time. In this section we will analyse the results obtained from the experiments depicted in figures 7 and 12. When comparing the two models retrieved from the CPU intensive 256MB profiling experiments, we observed that the execution times did not match. The execution time on the primary machine was 19 % higher compared to the secondary machine, even though the inputs of the experiment were the same. We assess several possible sources of this difference and discuss their influences.

6.1 Calibration Step

We investigate the calibration step as a potential source of this difference. It assumes that the used benchmark Linpack offers independent information about the CPU performance of a machine. Since it is widely used as a measure to quantify hardware performance, we consider it as a feasible and independent benchmark [DMBS79]. In the selection process, we looked at the performance measurements of both experiment machines and AWS Lambda instances at different CPU quotas and memory settings. We observed a linear increase in performance with the provisioned resources. The coefficient of determination R^2 was greater than 99.5% for all experiments conducted as shown in table 3. Therefore, we conclude that the benchmark represents a suitable and independent abstraction of the performance of a machine.

In the calibration step we executed the benchmark using a broad range of CPU quotas and memory settings. The quota only restricts the total CPU time that is available to a container. However, it does not limit the amount of virtual cores available to the container. Since Linpack is multi-threaded and uses one thread per physical core, the results can be influenced in an unpredictable way. In order to factor out these influences, we measured the single-threaded computing performance by pinning the benchmark container to one virtual core. On the local machine we used the resource limitation offered by Docker. On the provider side we also assign one CPU core to the benchmark process by utilising the `taskset` command[17]. It changes the affinity of a process to a set of virtual cores, in our case one. Because the benchmarking processes in total are limited to a fixed amount of CPU time in both the single- and multi-threaded experiment, we expect to obtain the same results. However, we found that the single-threaded computing performance differs from the multi-threaded version. On the primary machine we observed a decrease in measured performance of about 30%, whereas on AWS Lambda and on the secondary machine, we observed a decrease of about 5% to 8%. Since our CPU intensive simulation container is single-threaded, we repeated the experiment using the new performance model. While the model built for the secondary machine only changed marginally, the simulation container on the primary machine on average ran 2.5s faster, reducing the difference of execution times to 11%.

[17] Man page of `taskset`: `https://linux.die.net/man/1/taskset`

6.2 Influences of the Java Virtual Machine

Subsequently, the performance overhead of the JVM was investigated. As the initialisation of the JVM is mostly a CPU-bound process, we expect its performance to scale linearly with the provisioned CPU resources. In other words, we expect the initialisation time to be indirectly proportional to the container CPU quota. To assess this behaviour, we initialised the simulation container with an empty workload and measured the execution times using CPU quotas ranging from 10% to 100%. In the experiment, the observed execution times on average ranged from 210ms to 3580ms for the highest and lowest resource settings. As expected, the overhead remained linear and only contained small deviations. We suspect that the nevertheless present non-linearity stems from Java initialisation not being entirely CPU bound. While this overhead is relatively small and thus not the sole cause of the model discrepancy, future work must include factoring out the differences caused by the JVM. There is also a need to explore a way to minimise the overhead caused by the JVM initialisation.

Considering that the overhead still is higher than the non-linear deviation, we reassessed the benchmark as a possible sources of the observed offset. To factor out the overhead introduced by Java, we replaced our simulation container with a different CPU intensive task. We choose the CPU-bound version of the Sysbench benchmark[18]. Consuming only little memory, it calculates a set of prime numbers and offers, similar to Linpack, the hardware independent metric actions per second. We repeated the experiment with the new task and observed a considerable improvement in the accuracy of the mapping. The observed difference of measured execution times is 8%.

6.3 Resource Isolation

The limited resource isolation can also be a cause of the observed difference. As the three tasks Linpack, Fibonacci and Sysbench are all CPU-bound tasks with an ideally hardware independent metric, we are able to compare them. Abstracting from the provider performance model, we conducted an experiment analysing the computing performance of the three CPU intensive tasks on the two platforms. The CPU resources assigned to a container were confined to quotas ranging from 10% to 100%. From the measurements, we rebuilt the linear regression models describing the correlation between assigned resources and resulting performance. Because of the different performance units used, the models resulting from the combination of three tasks and two platforms cannot be compared directly. Thus, we scaled the three models of the secondary machine relative to their primary machine counterparts, which serve as a reference. We expect all three relative CPU models, each representing a portion of the reference model, to be the identical. Figuratively speaking, we predict the performance of the weaker hardware to be some percentage of the more powerful hardware for all of the three CPU-bound tasks. Smaller deviations can be expected especially for the Fibonacci and Sysbench tasks, as they are not explicitly designed to be hardware independent.

However, we noticed that this assumption does not hold for our experiment setup, as can be observed in figure 8. In the figure, the grey line represents the three performance models of the primary machine and serves as reference for the three relative models measured

[18]Sysbench benchmark: `https://github.com/akopytov/sysbench`

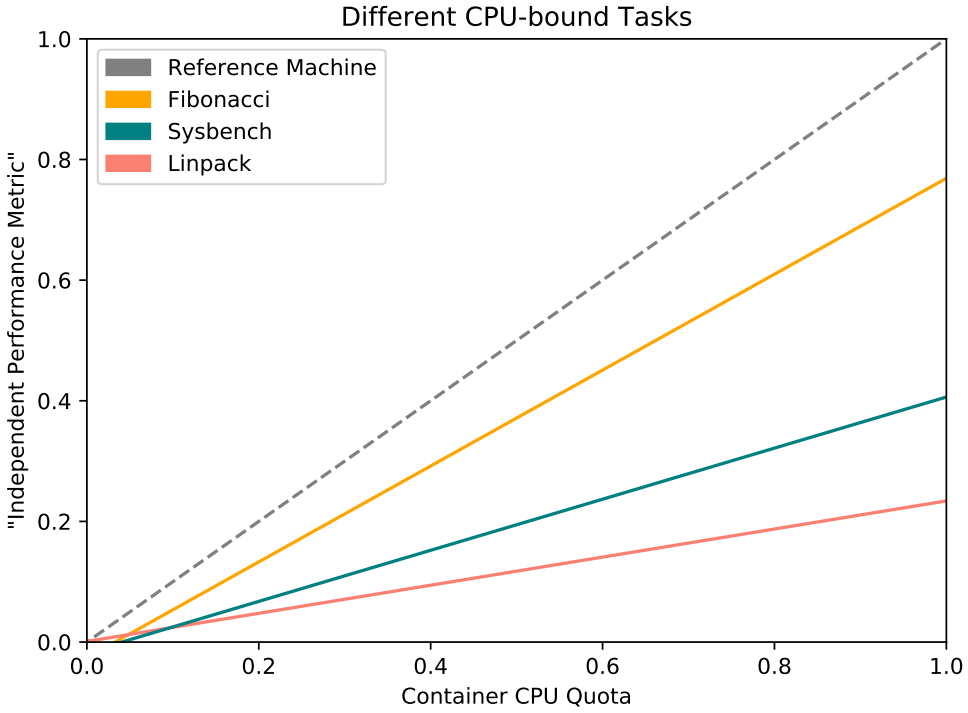


Figure 8: Performance Model Comparison of Linpack, Fibonacci, and Sysbench. The three performance models ($R^2 > 99.5\%$) represent the secondary machine's performance scaled to the models obtained from the reference machine.

on the secondary machine. We noticed that the Linpack performance model exhibits lower performance compared to the Fibonacci model. Under the assumption that Linpack is used as a calibration step, the Fibonacci task consequently performs better on the secondary machine. This also confirms our initial observation in figure 7, where we noticed differences among the two platforms. This experiment shows that the three tasks perform very differently on diverse hardware. Contrary to our expectations, the observed performances are not proportional to each other among the three CPU intensive tasks. The figure also explains the improved results achieved when we replaced the Fibonacci task for the Sysbench task. Assuming that Linpack is hardware independent, it shows that the Sysbench benchmark is more hardware independent than the Fibonacci task. As the experiment was only executed on two machines, it provides only limited information about the hardware dependence of the various tasks.

We suspect that these differences in the hardware dependence stem from the optimisations offered by different CPUs. While an independent mapping of performance is feasible, the cloud function itself can in isolation exhibit different performance characteristics on different machines due to other factors. Although it would make for a more realistic mapping, abstracting from these deviations is hard. Mapping the provider environment to local hardware is additionally impeded by limited operational visibility of cloud providers. For a correct hardware independent mapping of a cloud function, both the calibration step and the cloud function must ideally be hardware independent. Alternatively, the resources assigned to a container would be controlled more accurately resulting in a higher level of isolation. For this purpose, the calibration step would need to be extended to also consider different types of hardware optimisations. However, this would further complicate the experiment setup, where dependencies are less defined.

Apart from to the differences in hardware architectures, we also suspect that the Fibonacci task is more dependent on memory. The recursive implementation necessitates a large number of operations on the stack. Consequently, the primary machine will not benefit as much from its faster processor. Increasing the resource isolation further by also considering the memory speed of a machine is therefore crucial for a more realistic simulation.

To confirm the two assumption, a larger scale experiment on different hardware will reveal more information about the relative measurements. For this purpose, further benchmark algorithms may also be considered.

6.4 Comparison with AWS Lambda

In order to measure to what extent the local simulation reflects the provider environment, we additionally deployed the simulation cloud function to AWS Lambda. We ran the CPU intensive workload 100 times and compared it with the local measurements of the primary machine. The results of this experiment can be seen in figure 9, in which the execution times running the CPU intensive workload are compared with each other. In this experiment we only record the duration of the function executions, since the memory consumption is not as relevant and fine-grained CPU utilisation is not available to us. We observed that the execution times of each mapping did not match. The 2048MB mapping was slower on the simulated machine by 48.7%, whereas the 256MB mapping was faster by 32.3%. We suspect that the primary reason for this difference was the aforementioned hardware dependence of our experiment function discussed in section 6.3.

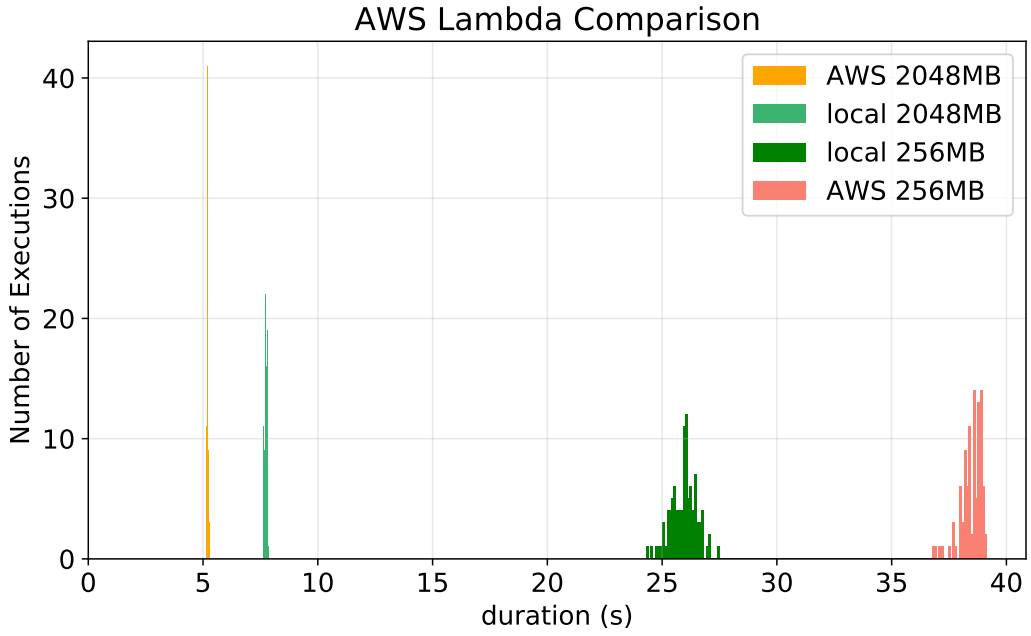


Figure 9: Comparison between execution durations on AWS Lambda and the Primary Machine running the CPU intensive workload. Instances are using 256MB and 2048MB as well as respective mappings ($n = 4 \times 100$).

It is apparent that the two pairs of measurements are not proportional to each other. This is the result of the local performance model intercept of -9.9, shown in table 3. While the reason for the intercept is unclear and requires further investigation, we assume that it stems from a non linear overhead imposed by the benchmark implementation. We repeated the experiment on the local machine using the performance models without the intercepts. The results did not improve as we observed average execution times of 122s for the 256MB setting. We also explain this observation with the non-linear overhead present in the cloud function.

We were able to weaken Eivy's claim that profiling a cloud function is "only possible on provider" by showing high accuracy of mapping the Sysbench task as an example cloud function[Eiv17, p. 9]. The examined hardware dependence of the Fibonacci task resulted in a higher discrepancy of measurements. It is therefore apparent that further steps are necessary to improve the isolation of resources.

7 Discussion

In our local profiling approach, the cloud function is run in isolation on the developer's machine. At the same time, cloud providers also use commodity hardware to offer their FaaS computing models to their consumers. While being based on similar hardware is an advantageous precondition to simulating another infrastructure, it does not imply that a realistic mapping is always possible without introducing side effects. We will discuss decisions made in the approach and highlight threats that might restrict the machine independence complementary to the results evaluation.

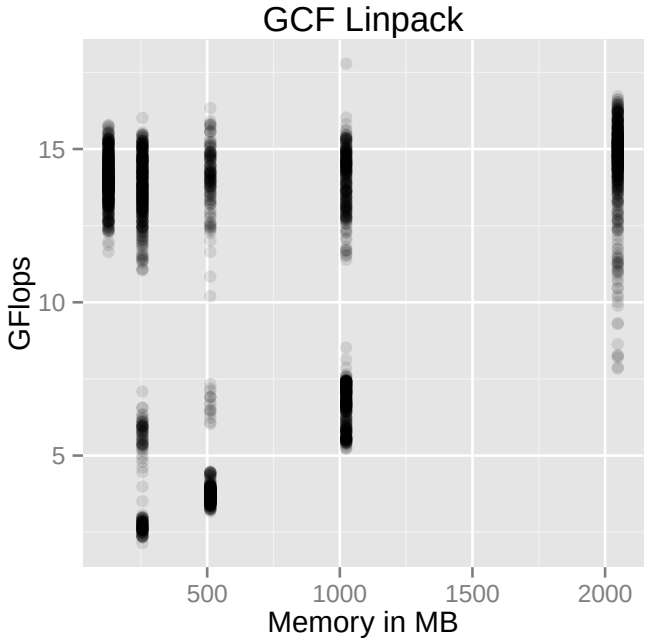


Figure 10: Linpack performance measured on GCP [MFGZ17].

Our proposed mapping from the provider to the local execution environment requires the association of settings regarding computing resource with a hardware independent metric quantifying the CPU performance. Both provider and local performance models need to define a form of correlation between the parameters and the CPU performance measured by this metric. We quantified this correlation on AWS Lambda using a linear regression model. Whereas on the local side, we have shown that it is possible to limit the CPU performance in a controlled way as shown in figure 4. This implies that simulation of different CPU capabilities is feasible. While we were able to model the correlation between memory setting and CPU performance on AWS Lambda, this relationship may not apply to every provider. For example, Malawski et al. were unable to find any correlation between resource settings and CPU performance in Google's FaaS offerings [MFGZ17]. The measurements of the original experiment are redrawn in figure 10 and show an irregular and unpredictable performance on different memory settings. If no relationship between resources and performance can be found, the model can instead provide information about the lower and upper bounds of CPU performance. Due to this heterogeneity of FaaS, it is more difficult to model the CPU performance accurately. A factor restricting this process further is the different levels of operational visibility and control over the hardware used by providers.

Moreover, the presented provider performance model is based on the raw computing performance of a machine. Since the FaaS paradigm is evolving quickly, this can vary over time. In the short term, performance may change slightly depending on the day of the week due to different service workloads [WLZ+18]. In the long term, authors observed frequent updates and adjustments to the offered services [vEIST17]. We were also able to confirm the latter observation using figure 5. It shows that the provisioned computing resources in the rerun experiment where lower than in the initial experiment conducted in 2017. Therefore, it is necessary to recalculate the provider specific performance models at regular intervals. Ideally, this process would be automated to detect changes in performance models as they occur.

The calibration step allows us to quantify the resources available both in the cloud and on local hardware. Since we repeated the benchmark originally defined by Malawski et al., we can compare our results to theirs [MFGZ17]. Although the original raw result data is not publicly available at the time of our experiment, two important observations can be made. First and foremost, the CPU performance within each memory setting varies much more in the initial experiment. The upper bound of the observed performance is greater than the lower bound by a factor of two, compared to an average difference of only 9.5% percent in our repeated experiment. Furthermore, in the original experiment, clusters with similar GFLOPS performance measurements can be observed in the Linpack instances with memory settings greater than 800MB. While the greater sample size of 12,000 compared to our 975 will have significant impact, there are some supplementary explanations for this. One possible explanation is that the VMs assigned to execute cloud functions run on different hardware. Wang et al. found that AWS Lambda uses at least five types of VMs whose host uses different CPUs [WLZ+18]. They observed a similar heterogeneity in GCP and Microsoft Azure who are using at least five and nine types of VMs, respectively. In our experiment, all the Linpack function instances provisioned by AWS ran on one type of VM using the Ivy Bridge Architecture at 3.2GHz, indicated by the CPU model name. The use of only one type of VM might be the reason for the lower variance in our results compared to the initial experiment. However, because of the small sample size of only 975 Linpack executions running on at least 66 distinct VMs, we cannot make any significant statement about the hardware used by AWS. Also, the number of VMs must be interpreted with precaution, since the used heuristic [LRC+18], which identifies unique VMs using their boot times, was shown to be unreliable [WLZ+18]. Since multiple VMs might have the same start time, this number can be slightly higher than 66. To either confirm or reject the latter presumption, a larger scale experiment with deployments to multiple regions must be conducted to reveal more insights.

Our local approach further abstracts from the provider execution environment by introducing assumptions that do not hold for FaaS infrastructure. For example, we expect the cloud function to be defined as a container image and ready to run with the consistent overhead imposed by the container initialisation. On the provider side, this is not the case, as they reuse containers and VMs to reduce the resource provisioning overhead [LRC+18]. By keeping used container instances warm for possible subsequent invocations, providers strike a balance between efficient resource management and the ability to scale vertically. This introduces side effects, like the cold start, which are not directly represented in our model. While this abstraction simplifies the profiling methodology, it further abstracts from the architecture used in the cloud. The model therefore can only be interpreted in isolation and on the function level.

The reproducibility is important for the validity of the results. We therefore focused on automating the profiling methodology as much as possible. Although this does not directly change the results, it comes with two important advantages. Firstly, since it refrains from manual steps like ad-hoc workflows and bash scripts, it makes the approach less error prone. Secondly and more importantly, researchers can rerun the experiment on their machines enabling a more objective evaluation [JMM+15]. With the aim of a more automated process, we included every parameter required for the experiment in a machine readable metadata file. Since the performance model depends on the provider calibration, the presence of the calibration function is a prerequisite to profiling. Apart from this exception, the experiment is repeatable on different hardware given that it supports the container technology.

The Java based resource load simulation image has several drawbacks. Due to the unpredictable nature of the JVM garbage collector, the repeatability of memory access simulation is limited. In section 6.2, we also observed a high and to some degree non-linear performance overhead when initialising the JVM using different CPU quotas. For example, the experiment simulating the 256MB memory setting on the secondary machine results in a high initialisation overhead. The corresponding local performance model assigns a CPU quota of 0.195 to the container. Running an empty load pattern results in an execution time of 2.1 seconds. The experiment showed that about 75% is JVM overhead and 25% is from the initialisation of the workload. To more accurately test the mapping feasibility, the use of a lower level programming language like `C` might be helpful. This would allow us to more directly control the access to CPU and memory resources.

The isolation of resources is an important challenge for cloud providers, especially with regard to the FaaS model. The practice of multi-tenancy allows for seamless scaling at a low cost, but introduces resource contention problems, e.g., when "noisy neighbours" run on the same machine or even on the same VM. Therefore, resource isolation requires cloud providers to allocate hardware resources to different VMs and containers without introducing resource interferences. Recent literature showed that the coresidency of VMs and containers results in considerable and unpredictable service degradations [VZRS15]. While providers mostly refrain from allocating individual VMs to multiple consumers, coresident function instances still introduce resource interferences [WLZ+18]. The problem of resource isolation is also a considerable threat to the validity of our local profiling approach, as the performance of the cloud function may be influenced by other programs running on the host. The resource isolation provided by container technology does not guarantee the availability of resources. It can only control resource limitations and not guarantee for perfect resource isolation. The host as a potential "noisy neighbour" is therefore a threat to the validity. Also, understanding the different resource assignment policies deployed by cloud providers is crucial in understanding problems which are introduced by the lack of isolation.

8 Conclusion and Future Work

This thesis proposes a profiling approach which aims to independently model the resource consumption of a cloud function. In the developed prototype, a simulation container consumes different types of resources in a controlled fashion. The container, which has been assigned resources corresponding to a cloud provider setting, is then profiled in a non-intrusive manner. Since these assigned resources have been determined using an independent calibration step, the resulting model abstracts from the experiment hardware used. This section will answer the research questions defined in section 1.2, assess the relevance of the work and propose improvements and extensions to our approach.

8.1 Conclusion

We set out to investigate three research questions described in section 1.2, which we will discuss in the following. Throughout our work, we focused on maintaining a high reproducibility of the methodology. The goal of RQ1: Reproducibility was to be able to develop a reproducible and consistent profiling technique, which does not introduce disproportional operational overhead. Although this is a relevant prerequisite for any systems research, we pay particular attention to it. Container technology helped us to realise this goal by offering independent and self-contained instances which in addition match well with the actual FaaS execution environment. Apart from the automation provided by container technology, we introduced an experiment metadata file containing all relevant experiment parameters. This further improved the repeatability among multiple experiment machines. To demonstrate the reproducibility, we tested the prototype on two machines and showed that its repetition did not introduce overhead. The comparison between the two experiments was also important for RQ2: Model Resource Usage. In our proposed model, we described the resource consumption of the cloud function independent of the two experiment setups used. Ideally, the two results would be exactly the same given that the same experiment input was used and the model is theoretically independent of the hardware. As this was not the case, we interpreted and explained the small differences that nevertheless occurred. We were therefore able to answer RQ2: Model Resource Usage by assessing the resource isolation of various types of resources used. Since the model is not completely independent of the experimental hardware, further steps are required to achieve this abstraction. Supplementary to the second research question, we investigated RQ3: Provider Mapping. In order to gain more information about the available types of resources at cloud providers, we introduced a calibration step which measures the available resources on the provider side. Using the resulting provider and local performance models, we mapped the provider resources to the local machine. Due to the already mentioned limited ability to isolate resources, we were only able to answer this question partially.

8.2 Contribution

Several authors claim that both the performance and the functional correctness of cloud functions can only be adequately evaluated in the cloud itself [Eiv17, JSSS+19, BCC+17]. They state that too many uncertainties, uncontrolled influences and the heterogeneity of the cloud render a local profiling approach infeasible. Authors thus see the local approaches as an open problem [vEIST17]. At the same time the operational visibility is limited and diverse among cloud providers. Combined with further side effects, this means that the information offered by cloud providers does not reflect the resource demands of a cloud function in full isolation and required granularity. Additionally, the information obtained is specific to one provider and relevant only for a certain period of time. This situation is very unsatisfactory for cloud consumers. It requires the cloud function to be deployed before it can then be evaluated in the cloud. The cloud consumer is consequently expected to decide on one of the different FaaS offerings which are becoming more and more diverse and sophisticated. Due to their heterogeneity and rapidly changing service models, this decision is not easy and the consumer only has limited information at their disposal. While local testing approaches have been proposed, they mostly focus on the functional correctness of a cloud function rather than their resource demands[19]. However, its resource consumption and behaviour during its execution is just as interesting as the correctness. The resource demands of a function determine the execution time and thereby the billed execution time, availability, and performance. With the interest to optimise one or a combination of these characteristics according to the needs of the cloud consumer, the choice of a service provider therefore plays a decisive role.

The vendor lock-in is intensively discussed as an open issue and drawback to cloud computing, in particular for the serverless computing model [FGJ+09, BCC+17, RC17]. It is the result of the overhead introduced when migrating a cloud infrastructure from one provider to another. In other words, it quantifies the commitment to one cloud provider, when a decision is made for that provider over another. Since our approach does not require this upfront decision, we are mitigating the problems introduced by the vendor-lock.

This decision is limited by the availability of information and additionally hindered by the unpredictability of billing models. Our proposed offline approach aims to bridge this gap by offering a decision aid to cloud consumers. The results of the proposed model serve as a means of quantifying the decision for consumers. Based on the independent model of the cloud function and the billing models of cloud providers, the consumer can calculate which provider is most suitable and least expensive for the cloud function. Furthermore, it provides guidance when resource settings need to be made on provider side.

[19] Microsoft Azure offerings for local testing and debugging a cloud function:
`https://docs.microsoft.com/en-gb/azure/azure-functions/functions-develop-local`
Command line tool to locally run and test your node.js cloud function to AWS Lambda:
`https://www.npmjs.com/package/node-lambda`

8.3 Future Work

The model obtained from profiling a single cloud function can serve as a decision aid for individual cloud consumers. However, they do not only deploy single cloud functions. A service oriented architecture often consists of multiple interconnected cloud functions working together to provide a service. To better reflect this, the model might be integrated in a higher level version to also consider interlinked services. The "CostHat" model proposed by Leitner et al. estimates the total deployment cost of a set of cloud functions [LCS16]. In their approach, the deployment cost of a single function is estimated using previous function executions. Instead, the model can estimate the cost independently of a cloud provider by using our local approach. This way, the model can convey provider independent information and estimate the performance and cost of a complete deployment to one of several cloud providers.

We discussed the threats to resource isolation in both cloud and local environments. While the used hardware will always influence the performance, isolating resources as much as possible will improve an independent evaluation. One option is to replicate the serverless execution environment more accurately by also using lightweight VMs. The function under investigation would then run in a container on the VM along side the profiling instance. Alternative to building it from scratch, the methodology may extend already existing solutions like OpenFaaS[20]. While it reduces the resource contention problems and thereby increases the accuracy of the measurements, it would entail additional management overhead. By contrast, container technology allows for exclusive assignment of CPU cores to different tasks. This CPU pinning technique would on a lower level facilitate resource isolation. However, it requires that the cloud function container is assigned to one virtual CPU exclusively. This means that all other tasks must be assigned the remaining virtual CPU cores explicitly. Furthermore, container technology offers more fine grained memory resource isolation capabilities[21]. The memory reservation option allows a soft limit smaller than the hard limit, which is enforced when the host is running low on memory. Since we have not found any previous experiments on this detail, it would be interesting to find out whether cloud providers use any related technique. This would better explain some of the observed resource contention patterns. Also, limiting the memory access speed of a container will greatly improve the simulation.

We envision our profiling approach to be integrated into the development process of a cloud function with little overhead. For the cloud function to be profiled without user interaction, this would require further automation of the process. In a Continuous Integration/ Continuous Delivery approach, the cloud function would be profiled and its resource usage modelled when an update is committed to the development repository. It must be decided what event triggers the profiling process and how it is incorporated in the deployment of a cloud function. For example, the model describing the resource use of the function can be created upon every major release. In the delivery pipeline, the cloud function would first be profiled using previously defined exemplary function calls. Because of the changing service models, this step requires the automatic and regular generation of up to date provider performance models [WLZ+18]. Next the model describing the function's resource demands are calculated. Based on current billing models offered by

[20] OpenFaaS – Serverless Functions Made Simple: `https://github.com/openfaas/faas`

[21] Specify the resources of a Docker container:
`https://docs.docker.com/config/containers/resource_constraints/`

the providers, the most suitable platform is selected for deployment. Finally, the optimal resource setting is assessed and selected for the cloud function. In this approach, the developer is no longer faced with the decision of choosing the appropriate provider and resource setting.

Currently, data intensive tasks which require a lot of network bandwidth are not very common in FaaS [JSSS+19]. Although these use cases are important, FaaS is not well suited compared to other stateful service models [HFG+18]. In addition to the stateless property, this is mainly caused by several open challenges discussed in recent literature [HFG+18]. As network I/O will gain importance with the expansion of possible FaaS use cases, it is important to also consider it for our model [JSSS+19]. Seeing that network performance scales with the memory setting in some service models (e.g.AWS Lambda), we envision a similar approach for mapping this type of resource to the local machine. Handigol et al. showed the feasibility of locally isolating network I/O resources [HHJ+12]. They repeated 18 networking experiments using their extended container-based emulation approach and were able to obtain reproducible results from 16 of them. In analogy to the computing performance assessment, an model would independently quantify the network performance on different memory settings [KPL19]. Analogous to controlling the CPU performance by means of container quotas, the network resources would be isolated to match the characteristics at the provider.

In the future, we envision that all approaches and models discussed will lose relevance as cloud providers raise the level of abstraction and reinforce the serverless paradigm. Although the option to adjust resources manually remains available, the provider would ideally find the optimal setting without consumer interaction. However, we do not expect this improvement in the serverless paradigm in the near future. In particular, the interoperability challenge between cloud providers if ever solved will take time [vEIST17]. Therefore, it is crucial to establish an objective decision aid for cloud consumers, who are faced with decisions in the quickly evolving and heterogeneous FaaS environment.

References

[ABDDP13] Giuseppe Aceto, Alessio Botta, Walter De Donato, and Antonio Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013.

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.

[BA18] Timon Back and Vasilios Andrikopoulos. Using a microbenchmark to compare function as a service solutions. In *Proceedings of the European Conference on Service-Oriented and Cloud Computing*, pages 146–160. Springer, 2018.

[BCC+17] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[BLM19] Maxime Belair, Sylvie Laniepce, and Jean-Marc Menaud. Leveraging kernel security mechanisms to improve container security: a survey. In *Proceedings of the IWSECC 2019 - 2nd International Workshop on Security Engineering for Cloud Computing, Aug 2019*, 2019.

[CP17] Emiliano Casalicchio and Vanessa Perciballi. Measuring docker performance: What a mess!!! In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 11–16. ACM, 2017.

[CSL04] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 15–28, 2004.

[DMBS79] Jack J Dongarra, Cleve Barry Moler, James R Bunch, and Gilbert W Stewart. *LINPACK users' guide*. SIAM, 1979.

[Eiv17] Adam Eivy. Be wary of the economics of" serverless" cloud computing. *IEEE Cloud Computing*, 4(2):6–12, 2017.

[FBS12] Juliana Freire, Philippe Bonnet, and Dennis Shasha. Computational Reproducibility: State-of-the-Art, Challenges, and Database Research Opportunities. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, pages 593–596. ACM, 2012.

[FFRR15] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Proceedings of the 2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 171–172. IEEE, 2015.

[FGJ+09] Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, and Ion Stoica. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009, 2009.

[HFG+18] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[HHJ+12] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 253–264. ACM, 2012.

[HW18] Christopher B Hauser and Stefan Wesner. Reviewing cloud monitoring: towards cloud resource profiling. In *Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 678–685. IEEE, 2018.

[JMM+15] Ivo Jimenez, Carlos Maltzahn, Adam Moody, Kathryn Mohror, Jay Lofstead, Remzi Arpaci-Dusseau, and Andrea Arpaci-Dusseau. The Role of Container Technology in Reproducible Computer Systems Research. In *Proceedings of the 2015 IEEE International Conference on Cloud Engineering*, pages 379–385. IEEE, 2015.

[JSSS+19] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383*, 2019.

[JSW+17] Ivo Jimenez, Michael Sevilla, Noah Watkins, Carlos Maltzahn, Jay Lofstead, Kathryn Mohror, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. The popper convention: Making reproducible systems evaluation practical. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1561–1570. IEEE, 2017.

[KJ13] Tomas Kalibera and Richard Jones. Rigorous Benchmarking in Reasonable Time. In *Proceedings of the ACM SIGPLAN Notices*, volume 48, pages 63–74. ACM, 2013.

[KPL19] Jeongchul Kim, Jungae Park, and Kyungyong Lee. Network resource isolation in serverless cloud function service. In *Proceedings of the 2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 182–187. IEEE, 2019.

[LCS16] Philipp Leitner, Jürgen Cito, and Emanuel Stöckli. Modelling and managing deployment costs of microservice-based cloud applications. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pages 165–174. ACM, 2016.

[LRC+18] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless computing: An investigation of factors influencing microservice performance. In *Proceedings of the 2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 159–169. IEEE, 2018.

[Man19] Johannes Manner. Towards Performance and Cost Simulation in Function as a Service. In *Proceedings of the 11th Central European Workshop on Services and their Composition*, 2019.

[Mer14] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[MFGZ17] Maciej Malawski, Kamil Figiela, Adam Gajek, and Adam Zima. Benchmarking heterogeneous cloud functions. In *Proceedings of the European Conference on Parallel Processing*, pages 415–426. Springer, 2017.

[MG+11] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[MKW19] Johannes Manner, Stefan Kolb, and Guido Wirtz. Troubleshooting serverless functions: a combined monitoring and debugging approach. *SICS Software-Intensive Cyber-Physical Systems*, 34(2-3):99–104, 2019.

[MRF18] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 35(4):11, 2018.

[OSN+17] Fábio Oliveira, Sahil Suneja, Shripad Nadgowda, Priya Nagpurkar, and Canturk Isci. A cloud-native monitoring and analytics framework. Technical report, 2017.

[PCZJ18] Aidi Pi, Wei Chen, Xiaobo Zhou, and Mike Ji. Profiling Distributed Systems in Lightweight Virtualized Environments with Logs and Resource Metrics. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 168–179. ACM, 2018.

[RC17] Mike Roberts and John Chapin. *What is Serverless?* O'Reilly Media, Inc. CA, US, May 2017.

[RTM+10] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE micro*, 30(4):65–79, 2010.

[USR09] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in a shared internet hosting platform. *ACM Transactions on Internet Technology (TOIT)*, 9(1):1, 2009.

[vEIST17] Erwin van Eyk, Alexandru Iosup, Simon Seif, and Markus 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.

[VZRS15] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A placement vulnerability study in multi-tenant public clouds. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security 15)*, pages 913–928, 2015.

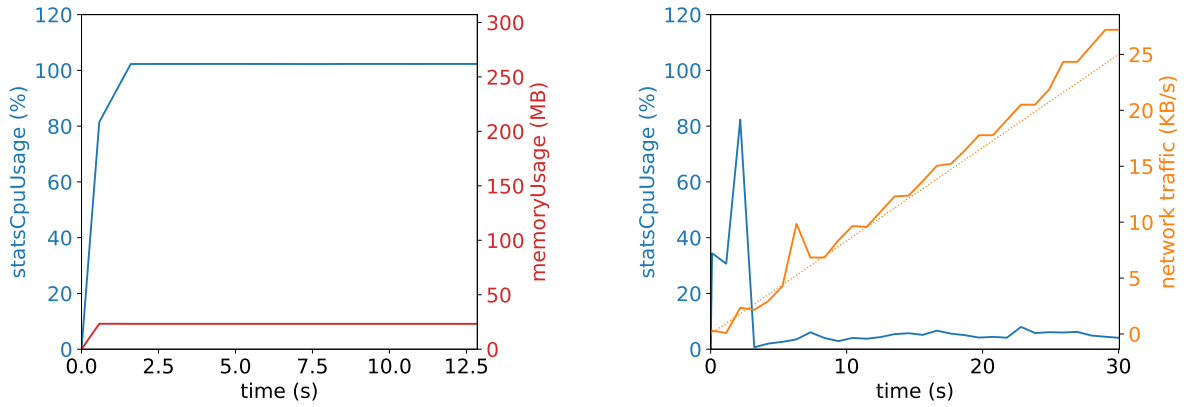
[WBW15] Rafael Weingärtner, Gabriel Beims Bräscher, and Carlos Becker Westphall. Cloud resource management: A survey on forecasting and profiling models. *Journal of Network and Computer Applications*, 47:99–106, 2015.

[WLZ+18] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Annual Technical Conference*, pages 133–146, 2018.

Appendix

Type	Primary Machine	Secondary Machine
Operating System	Manjaro Linux	Manjaro Linux
Kernel Version	Linux 5.2.8-1	Linux 4.19.62-1
Processor	2 × 4 Intel Xeon CPU E3-1231 v3 @3.40GHz	2 × 2 Intel i5 CPU M560 @2.67GHz
Memory	15.6 GB	7.6 GB
Docker version	19.03.1	19.03.1

Table 4: Experiment machines used. Unless otherwise stated, the primary machine was used.



(a) CPU load (absolute)

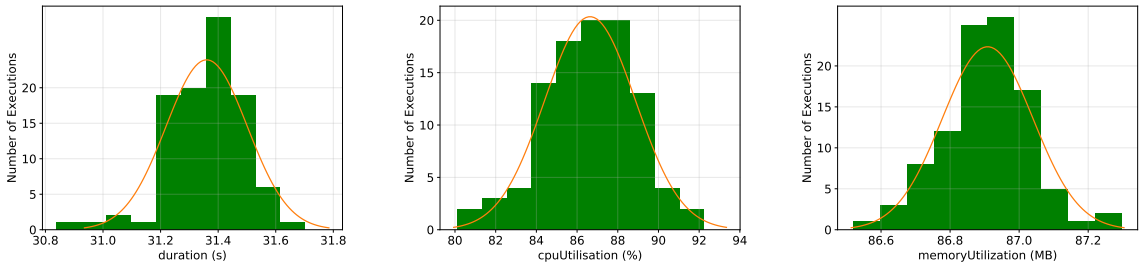
(b) Network I/O Simulation

Figure 11: Exemplary loads for CPU and memory loads.

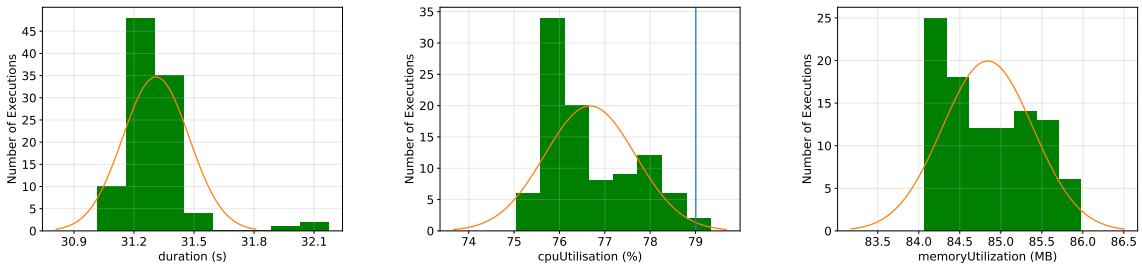
```
FROM gradle:jdk11 AS builder
WORKDIR /home/gradle/
COPY . .
RUN gradle :executor:build

FROM amazoncorretto:11
COPY --from=builder /home/gradle/executor/build/libs/* /app/
WORKDIR /app/
CMD [ "java -jar executor.jar ${JAVA_PARAMS}" ]
```

Listing 3: Dockerfile defining the load simulation image.



(a) Primary machine execution duration, CPU utilisation and maximum memory use model



(b) Secondary machine execution duration, CPU utilisation and memory use model

Figure 12: Model resulting from the mapping experiment running the CPU intensive load simulation comparing primary and secondary machine (n=100). The orange line represents the normal distribution, whereas the blue line indicates the 99% quantile for CPU and memory models.

```
1  {
2      "experimentName": "AWSPrimaryExperiment",
3      "quotas": [ 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0, 2.25,
           2.5, 2.75, 3.0, 3.25, 3.5, 3.75, 4.0 ],
4      "targetUrl": "https://6r69wtmag1.execute-api.eu-central-1.
           amazonaws.com/default/",
5      "apiKey": "<API KEY>",
6      "bucketName": "martin-linpack",
7      "memorySizes": [ 128, 256, 512, 768, 1024, 1280, 1536, 1792,
           2048, 2304, 2560, 2816, 3008 ],
8      "numberOfAWSExecutions": 75,
9      "simulatedMemory": [ 256, 2048 ],
10     "numberOfLoads": 100,
11     "loadPattern": {
12         "LOAD_TIME": "30000",
13         "CPU_FIBONACCI": "45",
14         "MEMORY_TO": "67108864" // only used for mixed workload
15     }
16 }
```

Listing 4: Experiment metadata file `experiment.json`. It shows the mixed workload calculating the 45^{th} value of the Fibonacci sequence and allocating 64MB of memory over 30 seconds. The CPU intensive task only calculates the Fibonacci value eliminating line 14.

Ich erkläre hiermit gemäß §17 Abs. 2 APO, dass ich die vorstehende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Bamberg, den 27.09.2019 Martin Endreß