BENCHMARKING RESOURCE MANAGEMENT FOR SERVERLESS COMPUTING

An Undergraduate Research Scholars Thesis

by

LOGAN KEIM

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Dilma Da Silva

May 2022

Major: Computer Science

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Logan Keim, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

ABSTRACT

Benchmarking Resource Management
for Serverless Computing

Logan Keim
Department of Computer Science and Engineering
Texas A&M University

Research Faculty Advisor: Dr. Dilma Da Silva
Department of Computer Science and Engineering
Texas A&M University

Serverless computing is a way in which users or companies can build and run applications and services without having to worry about acquiring or maintaining servers and their software stacks. This new technology is a significant innovation because server management incurs a large amount of overhead and can be very complex and difficult to work with. The serverless model also allows for fine-grain billing and demand resource allocation, allowing for better scalability and cost reduction. Academic researchers and industry practitioners agree that serverless computing is an amazing innovation, but it introduces new challenges. The algorithms and protocols currently deployed for virtual server optimization in traditional cloud computing environments are not able to simultaneously achieve low latency, high throughput, and fine-grained scalability while maintaining low cost for the cloud service providers. Furthermore, in the serverless computing paradigm, computation units (i.e., functions) are stateless. Applications, specified through function workflows, do not have control over specific states or their scheduling and placement, which can sometimes lead to significant

latency increases and some opportunities to optimize the usage of physical servers. Overcoming these challenges highlights some of the tension between giving programmers control and allowing providers to optimize automatically.

This research identifies some of the challenges in exploring new resource management approaches for serverless computing (more specifically, FaaS) as well as attempts to deal with one of these challenges. Our experimental approach includes the deployment of an open-source serverless function framework, OpenFaaS. We focus on faasd, a more lightweight variant of OpenFaaS. Faasd was chosen over the normal OpenFaaS due to not having the higher complexity and cost of Kubernetes.

As researchers in academia and industry develop new approaches for optimizing the usage of CPU, memory, and I/O for serverless platforms, the community needs to establish benchmark workloads for evaluating proposed methods. Several research groups have proposed benchmark suites in the last two years, and many others are still in development. A commonality among these benchmark tools is their complexity; for junior researchers without experience in the deployment of distributed systems, a lot of time and effort goes into deploying the benchmarking, hindering their progress in evaluating newly proposed ideas. In our work, we demonstrate that even well-regarded proposals still introduce deficiencies and deployment challenges, proposing that a simplified, constrained benchmark can be useful in preparing execution environments for the experimental evaluation with serverless services.

DEDICATION

Dedicated to my parents who always supported me through my education.

ACKNOWLEDGEMENTS

Contributors

I would like to thank my faculty advisor, Dr. Dilma Da Silva for their guidance and support throughout the course of this research.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

This work has been partially supported by NSF Award NSF-SPX- 1919181.

NOMENCLATURE

FaaS	Function as a Service
BaaS	Backend as a Service
AWS	Amazon Web Services
CLI	Command Line Interface
SeBS	Serverless Benchmark Suite
VM	Virtual Machine

1. INTRODUCTION

Serverless Computing is a relatively new innovation that allows a user or business to build and run applications without worrying about server management. This is especially useful because servers have an inherently complex programming and operating model. With serverless services, programmers can create applications using high-level abstractions that specify how the functions should run. Serverless computing also offers a pay-as-you-go cost model that heavily reduces expenses compared to the alternative of a reservation-based cost model. In addition to these qualities, serverless computing also has automatic, rapid, and unlimited scaling of resources to match the customer’s demand. Due to these beneficial qualities, many users are choosing to move to serverless computing services like Amazon Web Services (AWS) Lambda [2] or Microsoft Azure Functions [3]. Jonas et. al. [4], Schleier et. al. [5], and Castro et. al. [6] present detailed analyses of the challenges and opportunities of the serverless technology.

Two important services fall under the label “serverless computing:” Function-as-a-Service and Backend-as-a-Service [8]. Function-as-a-Service or FaaS [7] is a type of cloud computing service that allows customers to develop, run, and manage applications without having to maintain the infrastructure required to develop and launch an app. This means that the customer no longer has to worry about physical or virtual servers as well as the hosting process. The most common use of FaaS is building and launching microservice applications. Backend-as-a-Service is a cloud service model that lets developers outsource the behind-the-scenes aspects of a mobile or web application. This means that the developers only have to write and maintain the frontend portion of their application. Some of the common things this can be used for are user authentication, database management, cloud storage, and push notifications. Together, FaaS and BaaS make up serverless cloud computing.

Cloud functions allow applications to store an ephemeral state locally at each function. This is useful for caching and as working memory for the program. A serverless shared state can be saved in object storage or in key-value stores. The issue with these data services is that they incur overheads that have an adverse effect on low latency, low cost, high throughput, and fine-grained access, which are all possible with server-based data services. Approaches such as temporary data storage for analytics [9] and stateful cloud functions [10] that integrate caching attempt to address these challenges and provide consistency guarantees. Because cloud functions transfer the responsibilities of scheduling from the user-managed virtual machine to the cloud provider, several consequences arise. Because the user no longer has control over when each function runs, passing states between cloud functions requires a pass-through shared storage. This access adds a significant amount of latency. Users also cede control over where their functions run, thus also ceding the ability to do optimizations like sharing common inputs between tasks and combining outputs before sending them over the network. Overcoming these challenges highlights the tension between giving programmers control and allowing cloud providers to pursue optimizations that lead to the economy of scale achieved with server-based cloud services.

Both varieties of serverless computing can exhibit variable performance, which can cause issues if an application must meet strict performance guarantees. Part of this is due to serverless providers relying on statistical multiplexing to create the illusion of infinite resources. There always exists the chance that queueing delays can occur due to unfortunate timing.

1.1 Scheduling in Serverful Computing

The need and exploration of optimized approaches for scheduling has also happened when the traditional (i.e., virtual machine (VM)-based) cloud services were introduced. A first wave of optimizations deployed new algorithms for VM placement that pursued energy savings or hardware savings [11]. Then the academic and industry research community explored optimizations that improved application performance by taking into consideration communication patterns between VMs [12]. Recent investigations address optimization in memory usage [13] and placement of VM at the end of the network [14].

1.2 Experimental Evaluation of Scheduling Algorithms

The evaluation of system optimizations is a critical component of cloud computing research. It can be quite challenging to obtain experimental results that model system behavior [15]. There has been work based on models [16], regress test workloads [17], and simulators [18].

For serverless computing, there is no consensus yet on the appropriate toolsets or workload generators. Proposals in the literature include [19], [20], [21], [22] and [32].

2. BENCHMARKING METHODS

The evaluation of resource management algorithms requires obtaining experimental data that captures performance characteristics that highlights differences in application latency, throughput, or resource consumption. The complexity of cloud platforms (both server-based and serverless) is quite high, and the deployment of an experimental setup requires considerable expertise. This chapter documents some of the steps in this research in creating a benchmarking environment.

2.1 Creating a Development Environment

The first step to being able to modify a serverless function framework is to select one, download it, install it properly, and develop an understanding of how to use it and how it works. For the purposes of this project, we chose to use OpenFaaS, an open-source Serverless framework that can be used for free and is supported on multiple platforms [24]. Specifically, we selected to use faasd [24], a lightweight version of OpenFaaS that does not require the use and, therefore, the complexity cost of Kubernetes [23].

The next step is to decide where to install faasd and where to run all the functions using faasd. In order to emulate a real production environment, as well as to develop additional skills, we chose to use AWS [25] to run faasd and all of the serverless functions. Specifically,we use a Lightsail [26] instance running Linux, although there was some consideration to using an EC2 [27] instance for more flexibility. The reason we chose to use Lightsail over EC2 was its simple deployment, predictable cost model, and its ability to simulate an environment where using faasd can be more appropriate than just using OpenFaaS. This is due to faasd being a better choice in lower resource environments where Kubernetes could be to heavy weight to use.

In addition to installing all the services and systems required by faasd to get it up and running in a realistic environment, we will also need some way to measure the effectiveness of both the standard version of faasd right out of the box as well as to measure the effectiveness of the changes made. In order to actually measure how efficient faasd is, we will need to exercise the system using some variety of benchmarking workloads. Following the theme of using open-source software, we chose to use the Serverless Benchmark Suite, or SeBS for short. SeBS is a suite of diverse FaaS benchmarks that allows automated performance analysis of commercial and open-source serverless platforms [6]. One of the big benefits of using SeBS is that it not only works for big FaaS providers like AWS Lambda or Azure, but also works on custom Docker-based frameworks like faasd. Due to information in the literature describing SeBS as a fairly simple option that actually works to benchmark serverless services, we decided to adopt SeBS as the benchmarking tool in this research. With experimental data generated through benchmarking both the original faasd software stack and versions that incorporate new alternatives being explored, we can compare the results to assess if the scheduling modifications improve faasd or if faasd performs better as it is. It could also be the case that changing the scheduling algorithm makes no significant difference either way.

2.2 Developing an Understanding of faasd

After deciding on the tools and software to be set up for experimentation, the next step in the research model is to get everything set up and working in the target development environment. This task is actually more difficult than it initially sounds, due to the inherent complexity of deploying distributed systems [28], [29]. Setting up realistic experimental environments is one of the main hurdles that impede progress in advancing serverless platforms. Getting faasd (and its component faas-cli) installed on the server is quite trivial, achieved by just following the instructions on the faasd GitHub [33] and typing in the correct commands. Just installing faasd isn't necessarily enough, however, due to its dependencies. One may need to install a newer version of the Go [30] runtime system than the one installed. Identifying the appropriate version configuration can be quite time-consuming. Figuring out how to use faasd, on the other hand, is fairly easy due to the workshops and documentation [33] created by the openfaas creators. They do, however, require a small amount of translation though when using the faasd version, as the documentation is designed with OpenFaaS in mind. By thoroughly following the tutorials and doing some experimentation, we were able to develop a good understanding of how to use faasd.

The faasd deployment is only the initial step in the research process. Next, it is necessary to obtain an understanding of the faasd components. The available documentation focuses on the usage of faasd, not its internal design. After examining faasd code and figuring out how it works, we can proceed to actually modify its code and write our own resource management functions (e.g., implementing alternative scheduling algorithms) to change how faasd behaves. To do this, one has to learn the programming language go [30], since that is what faasd and the related libraries are all written in. This is, surprisingly, one of the easiest parts of the entire process, since Go is a programming language written by programmers, for programmers. This means that for experienced computer scientists with in-depth knowledge of other programming languages, it is fairly easy to read Go code and somewhat easier to write it.

2.3 Faasd Under the Hood

The next step towards changing the faasd scheduler was to figure out where (i.e., in which component) the scheduling is implemented. This is, once again, a more difficult task than it initially seems. After searching through the source code and adding output statements to trace the runtime, we narrowed down where the scheduling actually takes place. Surprisingly, this does not occur within the code of faasd itself. Such design is especially confusing given that the diagrams from the OpenFaaS documentation [24] show that the Fass-Provider component is quite important. What these diagrams don't show is that the FaaS-Provider does not actually do much in its own code. Instead, in the current implementation of faasd the scheduling of incoming function calls occurs within a library that faasd makes use of, containerd. Containerd is a Go library for running docker containers [31]. Containerd is quite a complex library, so creating an understanding of the library is challenging even for experienced software developers; for researchers, achieving the necessary understanding of the library can be incredibly difficult.

Using a combination of the source code and the containerd documentation, we discovered that the default faasd software uses a fairly simple first-in, first-out (FIFO) scheduling algorithm for each container it controls. Making this scheduler more efficient by changing its algorithm to something like a round-robin algorithm could make the return time (i.e., latency) more consistent for concurrent calls to the faasd function, thereby improving the faasd scalability. As with every scheduling algorithm, there are trade-offs between optimizing service time and providing fairness. Still, if enough calls are made, then changing the basic algorithm could significantly improve the scalability for larger applications. It could also be possible that there is not significant difference between our proposed approach's efficiency and that of containerd. To assess whether or not an algorithm is more efficient, benchmarking is necessary, as discussed in Chapter 1. We can run a benchmark suite (for example, SeBS) to exercise the original faasd and the proposed alternative algorithms. Through extensive experimental evaluation, we can comtrast the alternatives and determine which one ends up being more efficient.

2.4 Benchmarking

As stated above, our method utilizes SeBS in order to assess the performance of Faasd. SeBS has been published recently at a prestigious conference and is now starting to be used by the research community. As commonly is the case with new software packages, there a few different issues with trying to use a middleware like SeBS. Many of the packages that SeBS needs to run properly are not mentioned in the installation instructions. Many of the packages that fall under “standard Linux tools” are not actually already installed with standard Linux distributions and are also not used very often for other tools. Some google packages, like the Google Cloud Functions, especially need to be installed manually. Since most of these packages do not appear in the installation scripts or the installation instructions, the only way to actually figure out that they need to be installed is through trial and error. This way lets us find out what package is missing and then install it manually. Repeating this process many times over will eventually result in all of the packages being installed.

The incomplete description of package dependence is not the only roadblock with SeBS. While the documentation for SeBS is fairly sparse overall, it is especially lacking for running the FaaS benchmarking locally. A fair amount of experimentation was necessary to get some of the benchmarking tests running, but then the results of the tests did not seem to provide any relevant system behavior information. Because of the lack of relevant documentation for running local benchmarks, we moved into trying to benchmark one of the standard commercial serverless platforms like AWS Lambda or Microsoft Azure. After looking through the features and documentation of both, we chose to stay with AWS and use Lambda as the platform for benchmarking.

2.5 Simplifying the Benchmarking Experimentation

Serverless environments, in particular open-source ones, are new and the research community lacks experience with them. Due to the complexity of benchmarking tools like SeBS, we expected slow progress in utilizing SeBS. Difficulties in deploying such tools are likely to appear even within experienced researcher groups, but for people starting research in the field, the hurdle may seem unsurmountable. To address such challenges in using complex tools like the ones currently available, we changed our focus towards developing a simpler guide to benchmarking serverless computing.

The goal is to go back to the basics and create a simple "Hello World" type benchmark guide for AWS Lambda. Setting up stateless functions in Lambda is quite simple. The AWS Lambda infrastructure also offers a number of template functions and automatically tracks a few metrics whenever functions are invoked. These metrics are duration, billed duration, memory size, max memory used, and init (i.e., initialization) duration. Notably, init duration only appears for cold invocations, i.e., the function invocations that require the preparation of a new runtime environment to execute them. Consecutive invocations often end up having lower durations. This is due to the difference between "cold" invocations, the ones we are testing in our research, and "warm" invocations. Cold invocations occur when a function, in our case an AWS Lambda function, is called without any prior invocations. Proponents of serverless platforms argue that, in production environments, cold invocations are rare and not something developers need to be concerned about in terms of efficiency and function latency. Warm invocations occur when an AWS Lambda instance is already running when the Lambda function is invoked. The reason that these warm invocations are more common is that once an instance of AWS Lambda (or whatever serverless computing service you are using) is created, it will be kept active for a certain amount of time. If the application throughput is at a high enough level, then the instance may be kept almost perpetually active. Within this reasoning, cold invocations are uncommon because many applications in real production environments present an invocation pattern that is fairly consistent.

Since warm invocations may be more common in production environments, it seems counterintuitive that our work evaluates cold invocations. The main reason for choosing to focus on the cold invocations is similar to the thinking underlying many of the other design choices adopted in this study (e.g., faasd over OpenFaaS and AWS Lightsail over AWS EC2). We address serverless platforms for small-scale projects, like the ones that would use faasd, where cold invocations will be much more common and could possibly be the predominant type of invocation issued by the application.

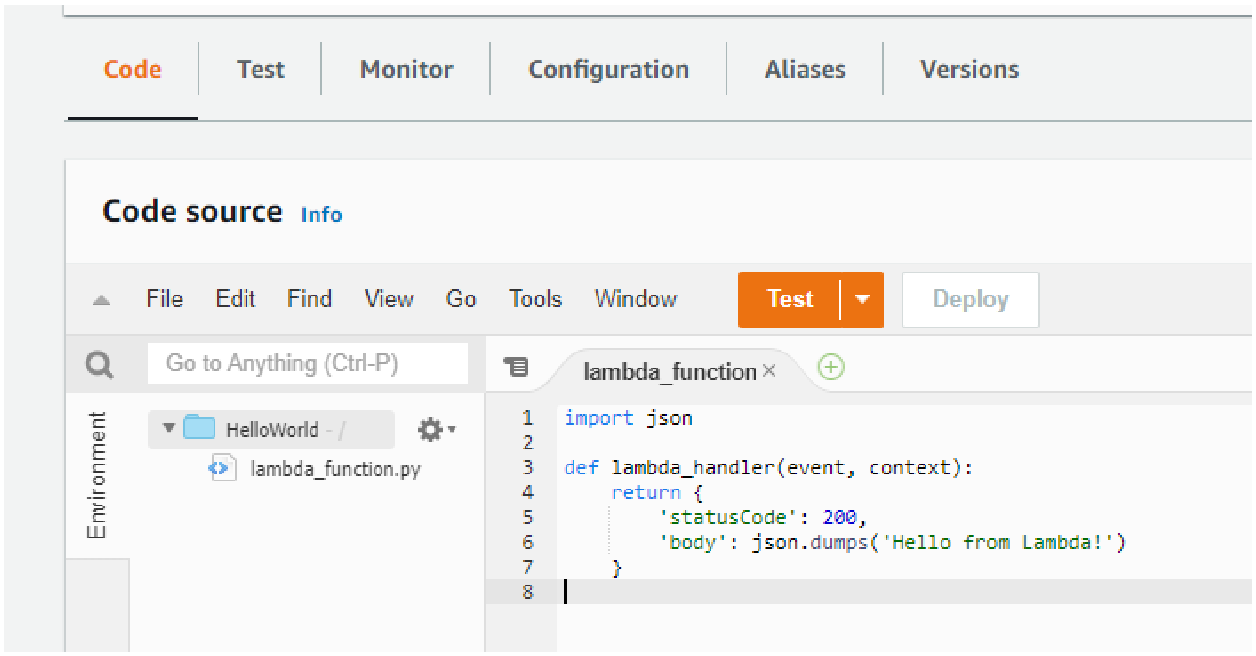
In order to demonstrate the basics of how simple benchmarking works, we set up a simple function in Python that returns “hello world”. Now that we have a simple function, just running it at various times of day can give some insight into the amount of time a function invocation will take during different times of day where traffic to AWS will be different. Having a simple function like this is useful because it eliminates package dependencies and won’t incur any real computation time. In general, using input/output operations introduce undesirable variability in execution time, and therefore should be avoided when benchmarking. But our goal is not to obtain performance data but to provide an initial entry point to experimenting with serverless benchmarking. The output operation makes it easier to confirm that the function is running as intended. Except for runtime variability, returning “Hello World” is trivial for an AWS Lambda function in terms of both time and computational complexity, and as an initial benchmarking step, it will not add any meaningful execution time to the invocation duration.

Since there is not much added time due to computational complexity, we can roughly assess how much time it takes for a cold invocation to start up an AWS Lambda instance to start the execution and return from a call. In addition to experimenting with such a simple function, it is also beneficial to perform the same assessments using a more complex function. There are a wide variety of "more complex" functions, but since the goal now is to make something simple, we will go with a function that is simple to implement but computationally intensive.

The function we decided to implement for this purpose is a standard matrix multiplication function. There are a couple of ways to do this, but given the purpose of this research, we implement a naïve matrix multiplication algorithm for simplicity. Since the naïve matrix multiplication algorithm is fairly trivial to implement and has a computational complexity that is high enough to matter for the assessment ($O(n^3)$), it is an appropriate choice. There are other ways to implement matrix multiplication (like the built-in functions for python), but these implementations are either more complex (straying away from the goal of simplicity) or vary by language of implementation. Since we aim at removing hurdles to this benchmarking method, we decided that sticking with the naïve method would be the simplest and most effective choice.

Now that we have decided upon our two functions of choice, the next step is to implement them in AWS Lambda. This task (unlike many of the previous steps), is actually fairly simple. The AWS Lambda service has better documentation than AWS Lightsail (likely due to Lightsail being less popular than its EC2 counterpart), making it fairly easy to diagnose any issues one might run into during use. AWS Lambda also has a fairly useful starter guide for making simple functions like the ones we will be using to conduct this research. When implementing these functions, there are a few different languages that can be used for the runtime of AWS Lambda functions. A few examples would be Go, Java, Node.js, Ruby, and

Python, as well as a few others, including custom bootstraps. For our functions, we chose to use Python in order to keep implementation simple and accessible to people lacking experience with more complex (and efficient) programming languages. Once the functions are implemented, testing them is easily achieved through the Lambda console. As shown in figure 2.1, there are specific features for the purpose of testing implemented Lambda functions One can also test the functions through specific triggers that will invoke a corresponding Lambda function. These triggers are very useful and are the main way these functions are actually used in practice, but for the purposes of this simplified benchmarking method, they are not necessary. Just using the test feature in the Lambda console returns the necessary information to make basic assessments about the runtime and invocation time of our simple function as well as our more complex function.



```python
import json

def lambda_handler(event, context):
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }

```

Figure 2.1: AWS Lambda Console [2]

The performance of function invocations also depends on the overall usage of physical servers and networks in the cloud provider’s datacenter, which varies over the hours and days.

Testing function invocations throughout the day will allow us to gather an idea of the time it takes to start up an AWS Lambda instance as well as the invocation time during different network traffic times. Presumably, there will be longer startup times during peaks in AWS traffic. These assessments show that our simplified method of benchmarking is an effective starting point for benchmarking various serverless systems.

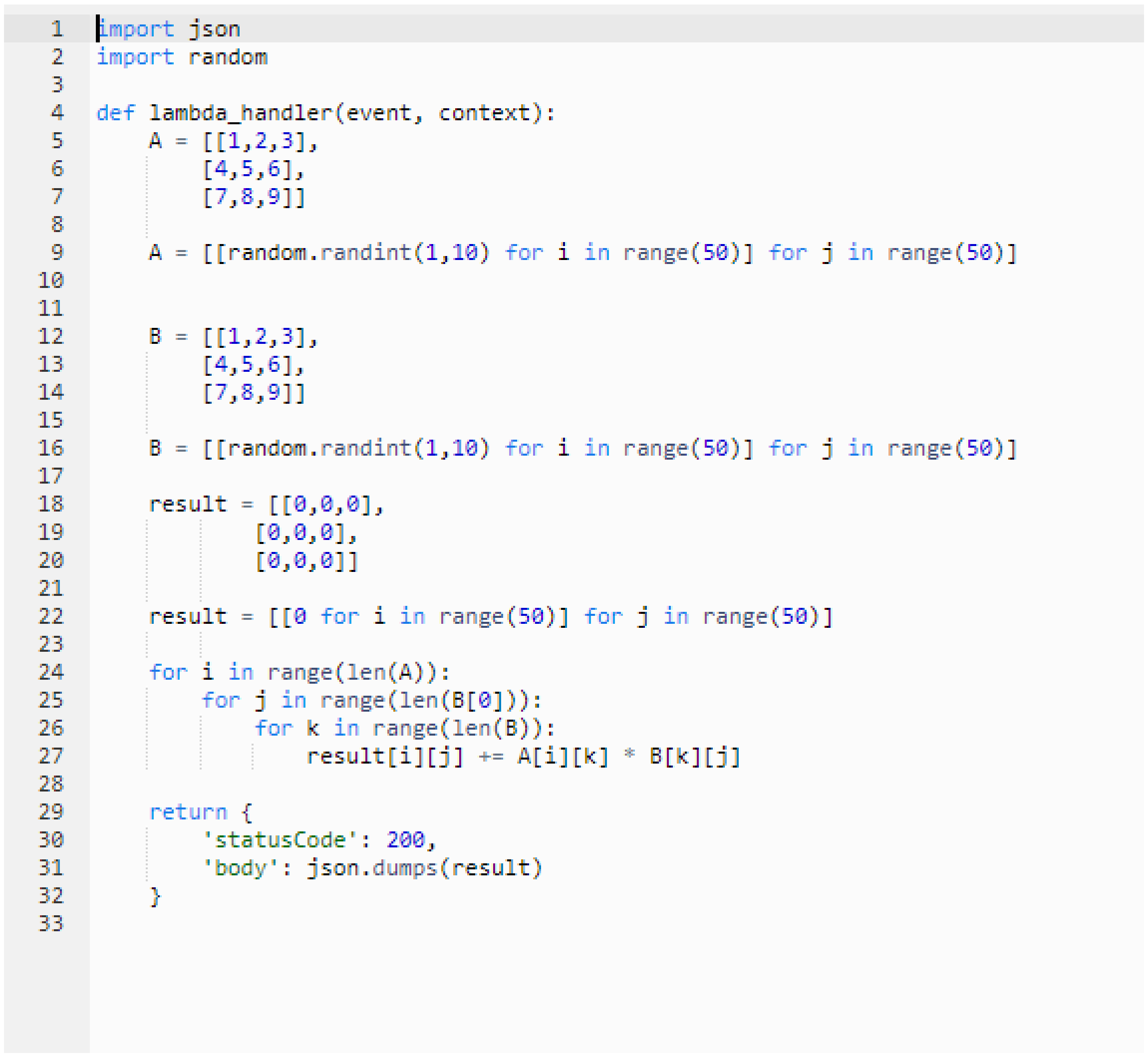
3. EXPERIMENTAL RESULTS

This chapter illustrates the usage of the proposed simplified benchmarking. We measure invocation time using simple functions on AWS Lambda, as described in the previous chapter.

In order to obtain the data presented here, we utilized the function test feature present in AWS Lambda (shown in Figure 2.1). By creating a test event using the default settings, we run the function and receive information about the invocation time (as well as some other information not presented here) from AWS. Once the test event is created for both the *matrix multiplication* function and the *hello world* function, all that remains is to exercise the functions to collect performance information.

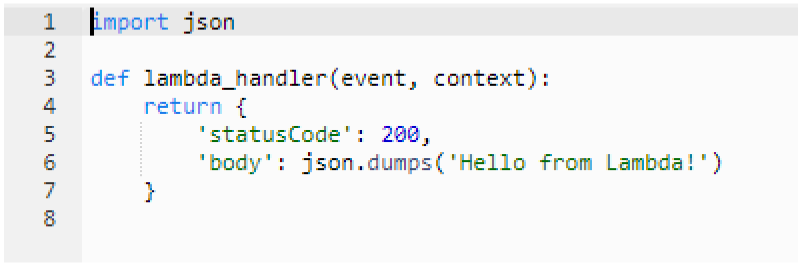
Cloud platforms host services for a large number of customers, which, in turn, may host many different applications. For example, the number of AWS users has been estimated to exceed 1 million [34], including companies from all economic sectors. In turn, each customer hosts many applications. The variety of hosted workloads, each with its own usage pattern, often results in significant disparities in data center loads across hours or days of the week. We chose to collect invocation time for functions triggered at intervals of 1 hour over the course of a day. The data collected and presented here was collected from invocations performed on a Saturday. This choice was made arbitrarily for ease of data collection.

Figure 3.1B depicts the code used for collecting invocation time for the simple “Hello World” function invocations over the course of a day. Figure 3.1A depicts the code used for collecting invocation time for the simple matrix multiplication function. The execution environment was an AWS Lambda instance. The Lambda instance used a 64-bit x86 architecture running Python 3.9. Given that AWS Lambda collects performance data with its data center infrastructure, latency perturbations on wide-area or local networks do not impact the experiments.



```python
import json
import random

def lambda_handler(event, context):
    A = [[1,2,3],
        [4,5,6],
        [7,8,9]]

    A = [[random.randint(1,10) for i in range(50)] for j in range(50)]


    B = [[1,2,3],
        [4,5,6],
        [7,8,9]]

    B = [[random.randint(1,10) for i in range(50)] for j in range(50)]

    result = [[0,0,0],
             [0,0,0],
             [0,0,0]]

    result = [[0 for i in range(50)] for j in range(50)]

    for i in range(len(A)):
        for j in range(len(B[0])):
            for k in range(len(B)):
                result[i][j] += A[i][k] * B[k][j]

    return {
        'statusCode': 200,
        'body': json.dumps(result)
    }

```

Figure 3.1A: Code specifying the Matrix Multiplication function.



```python
import json

def lambda_handler(event, context):
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda!')
    }

```

Figure 3.1B: Code specifying the Hello World function.

Tables 3.1 and 3.2 show the data collect in the experiment. With the goal of capturing invocation time for cold invocations (i.e., invocations for which a runtime environment had not been prepared yet by the serverless provider), we show the invocation time corresponding to one invocation only, instead of the common average or median of multiple invocations to reduce the impact of ephemeral events such as operating system noise or variations in access to the storage when retrieving the code for the function to be invoked. Figures 3.2 and 3.3 display the same data in a graphical form.

Table 3.1: Cold Invocation time for "Hello World"

Time of day	Time(ms)	Time of day	Time(ms)
12:00 AM	1.21	12:00 PM	1.51
1:00 AM	2.2	1:00 PM	1.01
2:00 AM	1.24	2:00 PM	1.3
3:00 AM	1.12	3:00 PM	1.21
4:00 AM	1.11	4:00 PM	1.23
5:00 AM	1.22	5:00 PM	1.11
6:00 AM	1.04	6:00 PM	1.05
7:00 AM	1.18	7:00 PM	1.1
8:00 AM	1.5	8:00 PM	1.33
9:00 AM	1.21	9:00 PM	1.06
10:00 AM	1.23	10:00 PM	1.08
11:00 AM	2.61	11:00 PM	1.08

Table 3.2: Cold Invocation for Matrix Multiplication

Time of day	Time(ms)	Time of day	Time(ms)
12:00 AM	517.89	12:00 PM	555.5
1:00 AM	503.86	1:00 PM	493.76
2:00 AM	536.84	2:00 PM	515.78
3:00 AM	490.82	3:00 PM	486.44
4:00 AM	499.89	4:00 PM	507.97
5:00 AM	536.95	5:00 PM	491.99
6:00 AM	532.3	6:00 PM	518.96
7:00 AM	556	7:00 PM	505.77
8:00 AM	506.96	8:00 PM	545
9:00 AM	544.5	9:00 PM	534.77
10:00 AM	536.08	10:00 PM	484.4
11:00 AM	532.04	11:00 PM	531.06

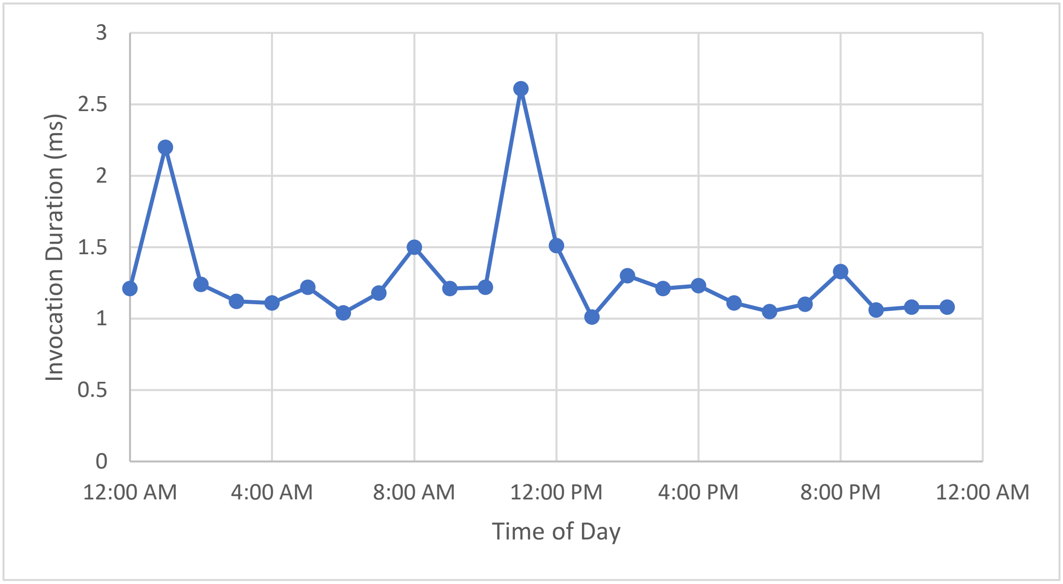


Figure 3.2: Invocations of "Hello World" over the course of a day

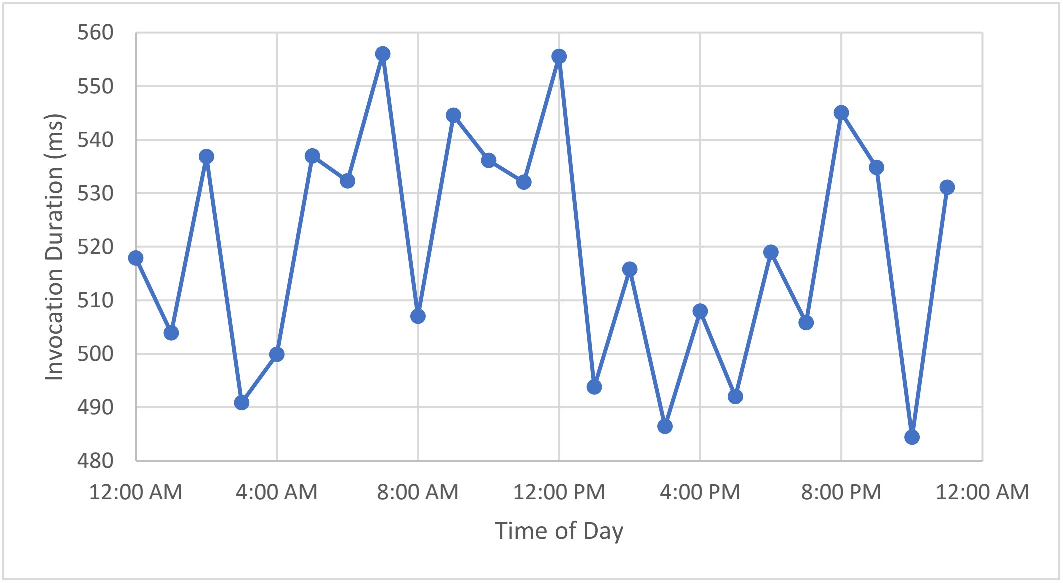


Figure 3.3: Invocations of Matrix Multiplication Function over the course of a day

The data collected in Table 3.1 (*Hello World* function) shows that there is significant variance (as large as 237%) in the amount of time it takes to perform a cold invocation during various times of the day. There are two noticeable spikes, one before 12 pm and the other just after 12 am. Many factors may contribute to such a large variance in serverless function invocations, and it is not possible to discern the cause without access to proprietary information on how the serverless platform is implemented. A common one is related to the elasticity of the platform: when there are no more containers available to host functions, additional servers may be provisioned, introducing a significant delay. Other possible causes may arise in the telemetry (i.e., monitoring) infrastructure at the provider due to possible network or storage congestion.

The data collected in Table 3.2, corresponding to the *Matrix Multiplication* function, exhibit much higher variance overall. In addition to the possible overhead sources identified with the *Hello World* experiment, the variance may also come from runtime aspects such as the level of co-tenancy in the platform. This highlights the complexity of assessing the performance of serverless functions, where the programmer has little control over the runtime.

The data collected here, however, don't track warm invocations. In order to have a better view of invocation time, it could be preferable to perform the same assessments as done already, but additionally repeatedly invoke the functions and take an average of the following invocations.

These tables and figures illustrate the value of the simplified benchmarking methods outlined in this research. Due to the simplicity and flexibility of this method, it can be adapted to other forms of serverless computing like Microsoft Azure or open-source versions like OpenFaaS. It also has a large amount of room for extension and expansion past the simple metrics assessed in our research.

4. CONCLUSION

Due to the many issues that need to be dealt with when exploring serverless computing platforms, creating a simplified framework for actually getting started and setting up an experimental environment for serverless computing is likely to be helpful for researchers aiming at optimizing resource usage in these platforms. Since the setup of software like OpenFaaS and SeBS is non-trivial, a guide to getting to some initial experiments may reduce the hurdles in the initial stages, accelerating the onboarding of new researchers (such as undergraduate researchers) in the field.

4.1 Benchmarking

The research conducted in this thesis work reported on the difficulties of working with serverless computing and the associated benchmarking services. Because serverless computing and its associated tools are often complex, they can be very hard to work with for those without several years of experience. Benchmarking tools like SeBS, which come from research groups, do not prioritize usability; as a result, they can be especially complicated and hard to work with. Since the documentation and instructions for these can often be lacking, just getting the tools running can be time-consuming and require acquiring unrelated skills such as package configuration. To address this difficulty, this research created a framework for starting the exploration and benchmarking of serverless platforms. Since benchmarking and experimental evaluations are critical parts of serverless research, the simplified method that we propose may be helpful in expediting the training of new researchers.

4.2 Future Work

In this thesis, we presented a simplified way of running experiments on serverless platforms with the goal of accelerating the initial training that researchers new to the area may need. The experimental plan we presented is fairly simplistic, therefore adding complexity to track additional features besides invocation duration is a natural progression.

An important aspect of benchmarking training is to understand what makes an experimental evaluation solid from a scientific point of view. Our work could be extended to include best practices in benchmarking software systems, as defined by Heiser's research group [35]. Our simple function invocation schemes can be an effective workload to learn about the intricacies of benchmarking.

Towards the goal of expediting future work, the next step could be to move from execution in a cloud provider towards local execution. For this purpose, our work could be extended to create a faasd-based package streamlined to run simplistic workloads.

Many benchmark platforms are currently being proposed. In this thesis, we examined a framework made available only six months ago. It is possible that the research community will reach a consensus over a specific system and accumulate experience that will remove the barriers we face when building an environment to run the benchmark. Even so, integrating our simplistic approach into a more complex framework could help new benchmark users by introducing simple workloads that can help identify configuration errors in a simpler environment.

REFERENCES

[1] Jia, Zhipeng, and Emmett Witchel. "Boki: Stateful Serverless Computing with Shared Logs." In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 691-707. 2021

[2] Amazon Web Services, "AWS Lambda Run code without thinking about servers or clusters." Available at https://aws.amazon.com/lambda/. Last visited in March 2022.

[3] Microsoft Azure, "Azure Functions documentation.", Available at https://docs.microsoft.com/en-us/azure/azure-functions/. Last visited in March 2022.

[4] Jonas, Eric, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar et al. "Cloud programming simplified: A berkeley view on serverless computing." *arXiv preprint arXiv:1902.03383* (2019).

[5] Schleier-Smith, Johann, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. "What serverless computing is and should become: The next phase of cloud computing." *Communications of the ACM* 64, no. 5 (2021): 76-84.

[6] Castro, Paul, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. "The rise of serverless computing." Communications of the ACM 62, no. 12 (2019): 44-54.

[7] Van Eyk, Erwin, Alexandru Iosup, Cristina L. Abad, Johannes Grohmann, and Simon Eismann. "A SPEC RG cloud group's vision on the performance challenges of FaaS cloud architectures." In Companion of the 2018 acm/spec international conference on performance engineering, pp. 21-24. 2018.

[8] Dudjak, Mario, and Goran Martinović. "An API-first methodology for designing a microservice-based Backend as a Service platform." Information Technology and Control 49, no. 2 (2020): 206-223.

[9] Pu, Qifan, Shivaram Venkataraman, and Ion Stoica. "Shuffling, fast and slow: Scalable analytics on serverless infrastructure." In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pp. 193-206. 2019.

[10] Shillaker, Simon, and Peter Pietzuch. "Faasm: Lightweight isolation for efficient stateful serverless computing." In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pp. 419-433. 2020.

[11] Silva Filho, Manoel C., Claudio C. Monteiro, Pedro RM Inácio, and Mário M. Freire. "Approaches for optimizing virtual machine placement and migration in cloud environments: A survey." Journal of Parallel and Distributed Computing 111 (2018): 222-250.

[12] Breitgand, David, and Amir Epstein. "SLA-aware placement of multi-virtual machine elastic services in compute clouds." In 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops, pp. 161-168. IEEE, 2011.

[13] Fuerst, Alexander, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. "Memory-harvesting VMs in cloud platforms." In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 583-594. 2022.

[14] Satyanarayanan, Mahadev. "The emergence of edge computing." Computer 50, no. 1 (2017): 30-39.

[15] Heiser, Gernote. "Systems Benchmarking Crimes." Available at https://www.cse.unsw.edu.au/~gernot/benchmarking-crimes.html. Last visited in March 2022.

[16] Hwang, Kai, Xiaoying Bai, Yue Shi, Muyang Li, Wen-Guang Chen, and Yongwei Wu. "Cloud performance modeling with benchmark evaluation of elastic scaling strategies." IEEE Transactions on parallel and distributed systems 27, no. 1 (2015): 130-143.

[17] Silva, Marcio, Michael R. Hines, Diego Gallo, Qi Liu, Kyung Dong Ryu, and Dilma Da Silva. "Cloudbench: Experiment automation for cloud environments." In 2013 IEEE International Conference on Cloud Engineering (IC2E), pp. 302-311. IEEE, 2013.

[18] Calheiros, Rodrigo N., Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms." Software: Practice and experience 41, no. 1 (2011): 23-50.

[19] Ustiugov, Dmitrii, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot.

"Benchmarking, analysis, and optimization of serverless function snapshots." In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 559-572. 2021.

[20] Somu, Nikhila, Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. "Panopticon: A comprehensive benchmarking tool for serverless applications." In 2020 International Conference on COMmunication Systems & NETworkS (COMSNETS), pp. 144-151. IEEE, 2020.

[21] Kim, Jeongchul, and Kyungyong Lee. "Functionbench: A suite of workloads for serverless cloud function service." In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 502-504. IEEE, 2019.

[22] Kuntsevich, Aleksandr, Pezhman Nasirifard, and Hans-Arno Jacobsen. "A distributed analysis and benchmarking framework for apache openwhisk serverless platform." In Proceedings of the 19th International Middleware Conference (Posters), pp. 3-4. 2018.

[23] Burns, Brendan, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. "Borg, omega, and kubernetes." *Communications of the ACM* 59, no. 5 (2016): 50-57.

[24] OpenFaaS Development Community. "faasd deployment - a lightweight & portable faas engine." Available at https://docs.openfaas.com/deployment/faasd/. Last visited in March 2022.

[25] Amazon. "AWS cloud computing services." Available at https://aws.amazon.com/. Last visited in March 2022.

[26] Amazon. "Amazon Lightsail" Available at https://aws.amazon.com/lightsail/ . Last visited in March 2022.

[27] Amazon. "Amazon EC2 Secure and resizable compute capacity for virtually any workload." Available at https://aws.amazon.com/ec2/. Last visited in March 2022.

[28] Meseguer, José. "Taming distributed system complexity through formal patterns." Science of Computer Programming 83 (2014): 3-34.

[29] Schantz, Richard E., and Douglas C. Schmidt. "Middleware for distributed systems: Evolving the common structure for network-centric applications." Encyclopedia of Software Engineering 1 (2001): 1-9.

[30] Meyerson, Jeff. "The go programming language." IEEE software 31, no. 5 (2014): 104-104.

[31] Cloud Computing Native Foundation. "containerd: An industry-standard container runtime with an emphasis on simplicity, robustness and portability.", Available at https://containerd.io/. Last visited in March 2022.

[32] M. Copik, “SPCL/serverless-benchmarks: Sebs: Serverless benchmarking suite for automatic performance analysis of FAAS platforms.,” *GitHub*. [Online]. Available: https://github.com/spcl/serverless-benchmarks/. [Accessed: 25-Jan-2022]

[33] A. Ellis, “Openfaas/workshop: Learn serverless for kubernetes with openfaas,” *GitHub*, 03-Apr-2019. [Online]. Available: https://github.com/openfaas/workshop/. [Accessed: 30-Mar-2022].

[34] Gillard, M. (2022, March 31). *Who's using Amazon Web Services? [2020 update]: Contino: Global transformation consultancy*. Contino. Retrieved April 3, 2022, from https://www.contino.io/insights/whos-using-aws/.

[35] van der Kouwe, E., Andriesse, D., Bos, H., Giuffrida, C., & Heiser, G. (2018, January 8). *Benchmarking crimes: An emerging threat in systems security*. arXiv.org. Retrieved April 3, 2022, from https://arxiv.org/abs/1801.02381/.