CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

AWS Lambda Language Performance

Bachelor of Science Thesis in Software Engineering and Management

MEHRSHAD HOSSEINI
OMID SAHRAGARD

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

A cloud benchmark on the Amazon Web Services Lambda platform

MEHRSHAD HOSSEINI
OMID SAHRAGARD

Supervisor: JOEL SCHEUNER
Examiner: RICHARD SVENSSON

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

AWS Lambda Language Performance

Mehrshad Hosseini
Software Engineering and Management, BSc
University of Gothenburg
Gothenburg, Sweden
gushosme@student.gu.se

Omid Sahragard
Software Engineering and Management, BSc
University of Gothenburg
Gothenburg, Sweden
gussahom@student.gu.se

***Abstract*—Cloud services are experiencing expansive growth, and the potential uses cases for its application in the IT sector is becoming increasingly widespread. This rapid growth is accompanied by a demand for performance which necessitates systematic benchmarking. The process of setting up a cloud-based benchmark is tedious and obstructive. Therefore, frequent benchmarks must be conducted in order to inform individuals from practitioners to hobbyists alike. Amazon Web Services is a major market leader in cloud computing which offers a Function as a Service platform named AWS Lambda. The following will conduct a benchmark on the Lambda platform and its supported languages: C#, Java, Node.js, and Python. The benchmark will examine the performance of the respective languages with relation to workload input size when configured to three different memory sizes: 128, 512, and 1024MB. The results presented reveal languages such as Java and C# consistently outperform the other languages with C# being the most performant when configured to 128 and 1024MB. All languages experienced a performance increase in tandem with increasing memory size.**

***Keywords*-cloud computing, AWS Lambda, cloud performance, benchmark testing, Function as a Service, FaaS.**

I. Introduction

Cloud computing has become widely adopted as an integral enterprising solution in the IT (Information Technology) market. It has experts projecting continued growth into 2021 [1]. The cloud platform market has become a profitable and competitive market led by technology giants such as Amazon, Google, and Microsoft [1]. This competitive nature allow cloud services to become more affordable and has led to greater adoption. Furthermore, one of the reasons for this growth is that both individuals and enterprises use cloud-based platforms due to their appeal of ease of use and convenience. Cloud-based service users run their applications or microservices in the cloud instead of hosting their applications using their own or rented infrastructure. There are various services which cloud computing platforms provide, however, this study has chosen to focus on a new emerging architecture called serverless. The incentive of a serverless architecture is to execute code or services without having to configure or maintain a server [2]. The focus therefore switches from Infrastructure as a Service (IaaS) to Function as a Service (FaaS). The cloud environment which we will be investigating is the Lambda service which is the FaaS solution provided by the current cloud market leader [3], Amazon Web Services (AWS).

The AWS Lambda platform offers a pay-per-request basis as the cost of a Lambda function execution is calculated based on the number of requests, runtime duration, and memory allocation [4]. In return, the cloud platform provides the compute power, source code storage, server configuration, deployment, networking, and scaling solutions to meet their clients' demands [5]. Therefore, it would be beneficial to conduct systematic benchmarking which explores which of the supported programming languages performs the quickest, thus having the lowest operational cost. Additionally, a market which is experiencing rapid growth may require continued re-evaluation to assess up to date adjustments such as new, custom hardware, and the latest cost optimisation techniques [6]. This will allow professional software engineers to hobbyist coders to obtain the best results through performance and cost.

There is insufficient, holistic systematic benchmarking aside from sprawled blog posts. The intention of this paper is to compare the programming language performance in different operational configurations on the AWS Lambda platform in order to contribute to the research space in a manner which facilitates reproducibility and validity.

II. Background

The following briefly expounds on the cloud computing nomenclature aforementioned, and explains the tools used to conduct the study.

A. The Appeal of Cloud Computing

There is currently no scientific consensus surrounding the definition of cloud computing, albeit there are similarities in terminology used by major providers to describe their cloud services.

”Cloud computing is the on-demand delivery of compute power, database storage, applications, and other IT resources through a cloud services platform via the internet with pay-as-you-go pricing.” - Amazon Web Services [7]

”In cloud computing, the capital investment in building and maintaining data centers is replaced by consuming IT resources as an elastic, utility-like service from a cloud “provider” (including storage, computing, networking, data processing and analytics, application development, machine learning, and even fully managed services).” - Google Cloud [8]

”Serverless computing is the abstraction of servers, infrastructure, and operating systems. When you build serverless apps you don’t need to provision and manage any servers, so you can take your mind off infrastructure concerns. Serverless computing is driven by the reaction to events and triggers happening in near-real-time—in the cloud. As a fully managed service, server management and capacity planning are invisible to the developer and billing is based just on resources consumed or the actual time your code is running.” - Microsoft Azure [9]

Cloud providers generally highlight four components in what makes their services appealing: [7]

- **Auto scaling.** Users will no longer have to estimate their capacity needs. As the number of requests increase, the cloud provider will provision the needed resources as deemed necessary. No bottlenecking.
- **Provision servers in a matter of seconds.** Users will no longer need to manage expensive, monolithic infrastructure. Users can now re-direct that capital investment into other sectors to benefit their customers.
- **Economies of Scale.** By aggregating many users in the cloud, providers are able to exploit this resource pooling to achieve lower prices for their users.
- **Pay-as-you-go pricing.** Only pay for the compute resources you consumed. Many enterprises only utilise approximately 20% of their IT infrastructure. This idle time is costly, now the capital spent on compute resources is efficiently utilised [10].

B. AWS Lambda

As aforementioned, Lambda is the Function as a Service platform provided by Amazon Web Services which allows users to deploy event-driven code, and is the sole benchmarking environment of this study.

1) Language Support: AWS Lambda currently supports the following languages: Node.js (JavaScript), Python, Java (Java 8 compatible), C# (.NET Core), and Go.

2) Tools for Deployment and Testing:

- **Serverless Framework.** The Serverless Framework, according to their documentation, is a “CLI tool that allows users to build & deploy auto-scaling, pay-per-execution, event-driven functions.” The framework currently supports 8 cloud providers where Amazon Web Services is one of them. CLI support allows the user to control everything from function creation to deployment to logging [11].
- **API Gateway.** Amazon API Gateway is a service which allows you to create, publish, maintain and monitor APIs. It will act as a gateway to run code on AWS Lambda or any other web application. The cost is determined by number of API calls and the amount of data consumed.
- **Artillery.** Artillery is a CLI load testing toolkit which offers a number of ways to test your services. There are quick tests where you can test a service through a URL with specific parameters such as how many requests and how many virtual users are sending those requests. Users can also run scripted tests which can achieve more complex behaviour.
- **CloudWatch.** Amazon provides its own monitoring service named CloudWatch which includes a wide array of tools to monitor the usage of your cloud resources. Tools can be used for gathering metrics, logging, and creating alarms.

3) AWS Lambda Resource Model: The Lambda resource model primarily revolves around the amount of memory allocated. Once function-specific configurations, for example, memory size, and environment variables are published, these settings are immutable for that version of the function [12]. AWS Lambda provisions compute power proportional to the memory allocation by applying the same ratio as a general purpose Amazon EC2 (Elastic Compute Cloud) instance type. A memory allocation of 256MB (megabyte) will net you twice the CPU power than if the function specified a 128MB allocation [13]. As aforementioned, the function-specific configurations are immutable, and must be re-configured if you wish to increase memory size. AWS Lambda allows increases in memory size in increments of 64MB, ranging from 128MB to 3008MB [14].

III. Related Work

The book titled “AWS Lambda: A Guide to Serverless Microservices” by Matthew Fuller [15] is to be used as a reference to Lambda concepts which will be supplemented by the official AWS Lambda documentation to ensure its relevance [16]. The two parameters this paper will examine in relation to language performance are: workload input size, and memory allocation. In the following section, we will overview the findings of related research and their approach to the solution domain.

Lynn et al. [1] discuss how FaaS has shown to increase cost-efficiencies by reducing configuration and management overheads, speed up the application and scale both ways as necessary depending on traffic. The most noteworthy property is the new business model where users of Lambda only pay on a per request basis [17], rather than paying a recurring fee regardless of use. The former business model, set up correctly, can provide a much more competitive price for users of FaaS platforms.

To run code in the cloud you must produce runnable artefacts that can be uploaded and deployed as a Lambda function. It has been empirically evaluated by Puripunpinyo & Samadzadeh that the size of a software artefact has an impact on the execution speed of the Lambda function as it will increase the duration of time required for it to be loaded into memory [18].

Cloud-based services must constantly juggle optimisation between power consumption and performance, therefore many platforms employ different optimisation techniques such as the cold start. A cold start is required if a function has gone through a period of inactivity where a new request will now require additional startup time, thus incurring additional overhead in terms of performance. Puripunpinyo & Samadzadeh

[18] demonstrate the benefits of reducing software artefact size in Java to decrease startup duration.

Singh et al. discusses how different performance evaluation approaches such as solving factorial or recursive calculations may affect language performance due to their language classification and low-level implementation. For example, a factorial computation strains the static memory allocation and this may have a more apparent effect on certain languages such as C++ and `C#` based on their memory management [19].

The research findings of Lynn et al. and Puripunpinyo & Samadzadeh note problems inherent to using Lambda. Lynn et al. mention optimisations techniques that can be done while Puripunpinyo & Samadzadeh lay out potential solutions to problems such as the deployment artefact size affecting the load and the cold start time. As demonstrated by reducing the deployment artefact [18], one can observe a noticeable increase in performance. Techniques to reduce the size of the artefact such as shrinking the artefact by transformation of the code, removing unnecessary code, obfuscation and minification. Compression of the artefact can also be done. A technique to reduce cold start times is to keep the containers warm by calling them frequently. This is a practice recommended by AWS [20]. The solution works well, however, the problem remains when the function needs to scale because the container must be reinitialized. Puripunpinyo & Samadzadeh note that for this case it is important to keep the artefact size as low as possible because this is the only factor that can reduce the cold start time. One of the main advantages of using serverless architectures is the freedom of not configuring any infrastructure. Even though creating functions on the AWS Lambda platform is relatively simple, it can be a hassle to manage a lot of them. As a result, the Serverless Framework [11] has been developed to make the deployment to FaaS platforms even easier. We will be using this framework to simplify and automate our deployment process. Also, Yan Cui discusses [21] Artillery.io [22] as a great toolkit for load testing, which can also be used to measure the performance of Lambda functions as previously discussed in the ”Tools for Deployment and Testing” sub-section of the background section.

IV. Research Methodology

This section will discuss the research questions, and the methodology employed in the study design. Lastly, it will detail the threats to validity.

A. Research Questions

The study consists of one primary research question (RQ1) and its derived sub-questions (SQ1-2).

RQ1: How do the supported programming languages perform on the AWS Lambda serverless platform?

SQ1: How do the different memory configurations affect the languages' performance?

SQ2: How does workload input size affect the performance?

B. Benchmark Design

The study has selected a quantitative research approach to explore the aforementioned research questions as it is the most applicable in observing the execution time of the Lambda functions. Cloud performance benchmarking is a tedious activity given the amount of preparation time needed to set up a benchmark for deployment. Therefore, in order to make it easier to work with Lambda and facilitate the benchmarking process, we have chosen to use the Serverless Framework [11] which provides the user with a number of tools to simplify the deployment process to FaaS platforms. In addition to using the Serverless Framework, we have developed a number of scripts to further ease our data collection. As there are variables that need to be adjusted, it is essential that scripts and tools function as fluid as possible to automate multiple facets of the study in order to negate potential error-prone tasks were it to be done manually. The variables that need to be adjusted accordingly are the following: the memory configuration for the function and the array size. The sorting algorithm used to sort the unsorted array has been extracted from Rosetta Code [23]. Rosetta Code is a programming chrestomathy archive which aims to provide similar solutions to the same task in different programming languages. Listings 1 and 2 provide a couple of examples of solutions given by Rosetta Code. This study will benchmark all the supported languages except Go due to time constraints.

```
  public static void main(String[] args)
    {
        int[] nums = {2,4,3,1,2};
        Arrays.sort(nums);
    }
}
```

Listing 1. Java Integer Array Sort Example

```
using System;
using System.Collections.Generic;

public class Program {
    static void Main() {
        int[] unsorted = {6,2,7,9};
        Array.Sort(unsorted);
    }
}
```

Listing 2. C# Integer Array Sort Example

The executable code is uploaded and deployed as a Lambda function by use of the Serverless Framework. The memory allocation is defined in the serverless.yml configuration file which is packaged with each function. The memory allocation value is then defined by an environment variable. By altering the environment variable and re-deploying the function, the new memory allocation value will be applied. The benchmarking process employs a script that we use to automatically build and deploy each function to our AWS Lambda account, this automation is key to saving time in a time-consuming task as the deployment process is repetitive throughout the study. Furthermore, as we are using a sole function per language, these functions consist of generating a random integer array with the desired array size along with their respective sorting algorithm extracted from Rosetta Code. We originally intended to create arrays of differing sizes where we would then insert the data into each function with the aim of having the same array data for each language using their respective sorting algorithm. The problem we faced was that this approach limits the array size to around 15,000 integers for some languages. A class or method can only make up a certain size which was easily exceeded with our intended array sizes. Due to time constraints, we chose to populate the arrays with random integers from 0 up to the desired array size. Our tests showed that this approach did not give us a major variance in results, although it is a confounding variable. The testing process is repeated with different memory allocation: 128, 512, and 1024MB. For each memory size, the benchmark proceeds with an array size of 0 and is incremented by 5,000 after each invocation until the array size reaches 1,000,000. This is the planned range due to a timeout issue concerning Amazon API Gateway as there is a limit for how large the array sizes could be. The reason for this is that we use the API Gateway to call our Lambda functions, where we would pass the desired array size as a parameter in the request. AWS has chosen to put a limit of 30 seconds on each request made to their API Gateway, which means that our array sizes could not exceed a million integers. It could work for the compiled languages, however, the interpreted languages perform slower for larger input sizes. Therefore, there was a need for a common array size ceiling where we could benchmark all languages to an upper bound where none of the Lambda functions would timeout.

Once Lambda functions are deployed, the Serverless Framework client returns a URL that you can use to call the function. We then use Artillery [22] to simulate a user request. In order to simplify and automate the testing process, a script was employed for this purpose. By running our main script, it runs four testing instances that call the URL of each language with the array size parameter starting from 0 and incrementing by 5000 until reaching 1,000,000 per the timeout limit restricted by the API Gateway. We then repeat this process with different memory configurations in order to conduct a quantitative data collection. Ultimately, this research will analyse the acquired data to conduct a comparison of the runtime performance of each language in different configurations. All tests were performed on the AWS Lambda (N.Virginia) region.

C. Benchmark Data and Analysis

Once the data collection process has been completed, the benchmark data is then exported from CloudWatch to a CSV (comma-separated values) format to then be manipulated in RStudio. This study employed a combination of RStudio [24] and CloudWatch for its statistical analysis and data visualisation. All software artefacts produced such as scripts, Lambda functions, and configuration files are available on a public GitHub repository [25].

D. Validity Evaluation

Studies which conduct quantitative analysis must strive to achieve a degree of validity in order to claim substantive arguments and ensure research quality.

1) Construct Validity: Threats regarding construct validity concern the relationship between theory and observation. Empirical studies with a high level of construct validity ensures that the measured variables are relevant to answering the proposed research questions. To ensure that the correct metrics were measured, this study made use of built-in monitoring and logging tools officially supported by AWS Lambda such as CloudWatch.

2) Internal Validity: Internal validity is the measurement of the casual relationship produced by a study with regard to the magnitude of confounding influences. The generation of random integers to be used as our main dataset will be a confounding variable. A confounding variable of the algorithms extracted from Rosetta Code may be that the solutions are not the most optimal, thus do not show the full capabilities of a language, however, these generic solutions are the ones currently accepted by the community.

3) External Validity: This form of validity concerns the research's capabilities to generalise and extend the study beyond the scope of the findings. All languages were tested against a generic integer array sorting algorithm which is a commonplace in the programming world, however, it may not be entirely representative of a language's overall performance.

V. Results and Discussion

The following section addresses the results of the benchmark which is arranged according to the research questions presented in the research methodology section.

1) How do the different memory configurations affect the languages' performance? **(SQ1)**

TABLE I
AWS Lambda Benchmark

Language	Memory Size (MB)	SD. (ms)	Mean (ms)	Median (ms)
Python	128	5457.40	8842.45	8909.79
Python	512	1625.73	2383.40	2449.63
Python	1024	652.36	942.99	895.73
Java	128	640.39	1051.88	966.50
Java	512	94.95	156.21	145.35
Java	1024	46.83	87.23	85.18
Node.js	128	4605.92	7584.17	7684.71
Node.js	512	1093.38	1669.75	1739.90
Node.js	1024	626.78	931.01	869.94
C#	128	446.38	703.68	691.26
C#	512	117.55	173.93	167.61
C#	1024	55.70	83.48	78.35

TABLE II
AWS LAMBDA BENCHMARK - PART 2

Language	Memory Size (MB)	Total (s)	Maximum (ms)	95th (ms)	99th (ms)
Python	128	5329	18432.25	16955.94	17915.35
Python	512	1430	5267.25	4867.08	5120.91
Python	1024	565	2100.53	1988.42	2087.64
Java	128	638	3343.51	2148.03	2673.10
Java	512	132	476.28	303.63	391.71
Java	1024	53	216.44	162.71	173.21
Node.js	128	4550	16459.30	14892.62	15780.43
Node.js	512	1371	3718.96	3382.19	3649.70
Node.js	1024	558	2207.72	1978.63	2069.95
C#	128	428	1611.12	1397.34	1521.31
C#	512	106	417.12	381.45	404.09
C#	1024	50	213.91	171.72	199.19

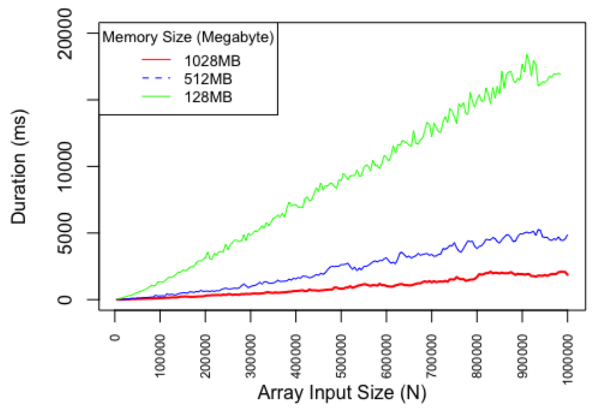

Fig. 1. Python Benchmark - 128, 512, 1024MB

TABLE III
PYTHON - PERCENTUAL REDUCTION IN EXECUTION TIME

Memory → Memory (MB)	Percentual Reduction in Execution Time
128 → 512	73%
128 → 1024	89%
512 → 1024	60%

The following can be extracted from the Python results regarding different memory sizes. There is a significant increase in performance in tandem with increasing memory size. Table 3 shows that a memory increase from 128 to 512MB results in a 73% reduction in duration, and a jump to 1024MB obtains an 89% reduction. A move from 512 to 1024MB resulted in a 60% reduction in invocation time. Furthermore, a memory size of 512 or 1024MB gives the user more consistent performance as demonstrated in Figure 1.

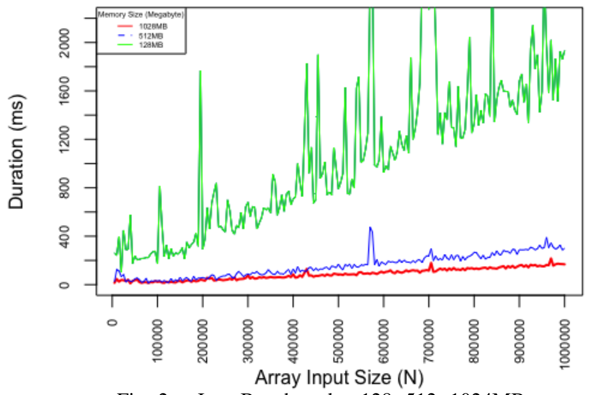

Fig. 2. Java Benchmark - 128, 512, 1024MB

TABLE IV
JAVA - PERCENTUAL REDUCTION IN EXECUTION TIME

Memory → Memory (MB)	Percentual Reduction in Execution Time
128 → 512	79%
128 → 1024	92%
512 → 1024	60%

The Java benchmark results find a preference to higher memory allocation. Table 4 enumerates a noticeable performance increase from 128 to 1024MB which resulted in a 92% reduction in duration. A large performance increase was also found when switching from 128 to 512MB which saw a 79% reduction. As seen in Figure 2, the memory size of 128MB resulted in a larger variance in performance when compared to its 512 and 1024MB counterparts.

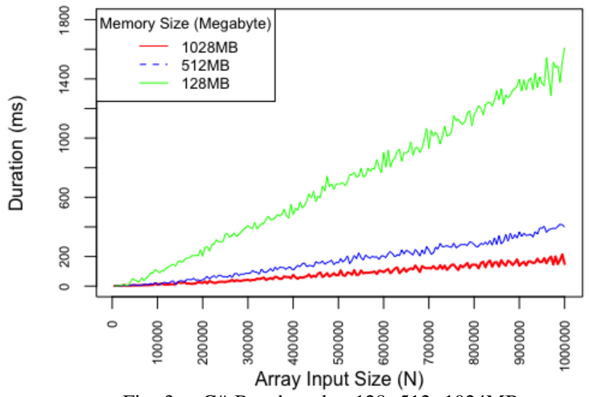

Fig. 3. C# Benchmark - 128, 512, 1024MB

TABLE V
C# - PERCENTUAL REDUCTION IN EXECUTION TIME

Memory → Memory (MB)	Percentual Reduction in Execution Time
128 → 512	75%
128 → 1024	88%
512 → 1024	53%

The C# benchmark resulted in similar performance improvements. Table 5 reveals that a configuration change from 128 to 512MB saw a 75% reduction. Moreover, 128 to 1024MB obtained a 88% change while a jump from 512 to 1024MB met diminishing results and improved by a lesser 53%.

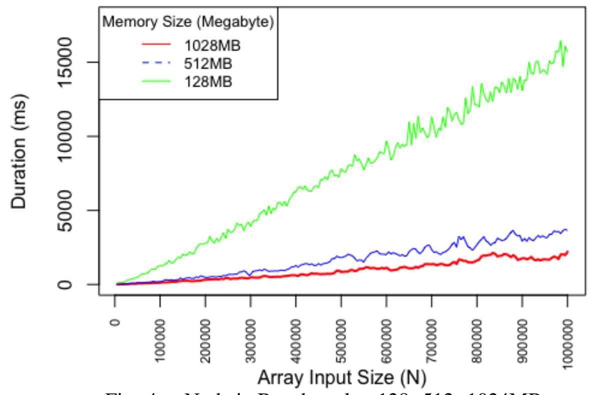

Fig. 4. Node.js Benchmark - 128, 512, 1024MB

TABLE VI
NODE.JS - PERCENTUAL REDUCTION IN EXECUTION TIME

Memory $\rightarrow$ Memory (MB)	Percentual Reduction in Execution Time
128 $\rightarrow$ 512	70%
128 $\rightarrow$ 1024	88%
512 $\rightarrow$ 1024	59%

Following the Node.js tests as provided by Figure 4 and Table 6, a reduction of 75% was realised when switching from 128 to 512MB. A larger reduction of 88% can also be found when increasing the memory size from 128 to 1024MB. 512 $\rightarrow$ 1024 resulted in a 53% reduction.
Overall, all languages experienced performance improvements when configuring a larger memory size as expected and stated by Amazon Web Services [12]. However, languages such as Java reaped the greatest benefit with higher memory allocation both in performance and consistency. A Java function memory allocation of 128MB resulted in worse, inconsistent performance in comparison to other languages which did not experience such a large variance in performance while having the same memory size.

2) How does workload input size affect the performance? **(SQ2)**

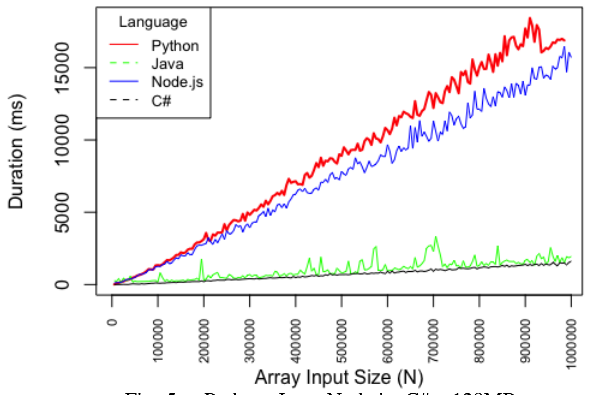

Fig. 5. Python, Java, Node.js, C# - 128MB

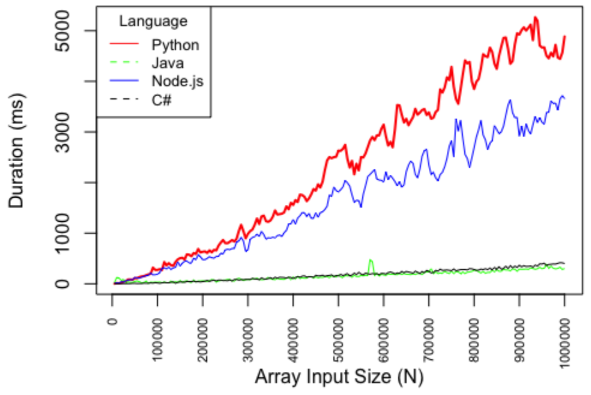

Fig. 6. Python, Java, Node.js, C# - 512MB

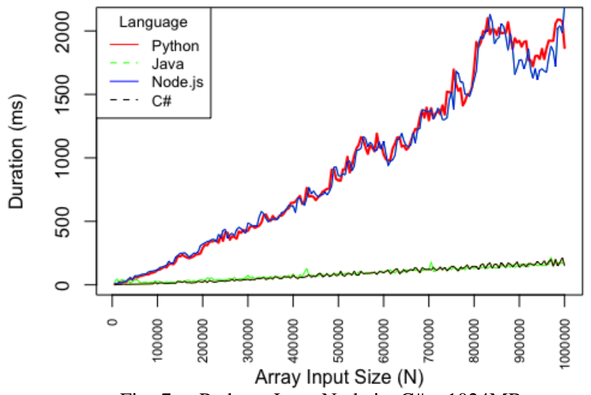

Fig. 7. Python, Java, Node.js, C# - 1024MB

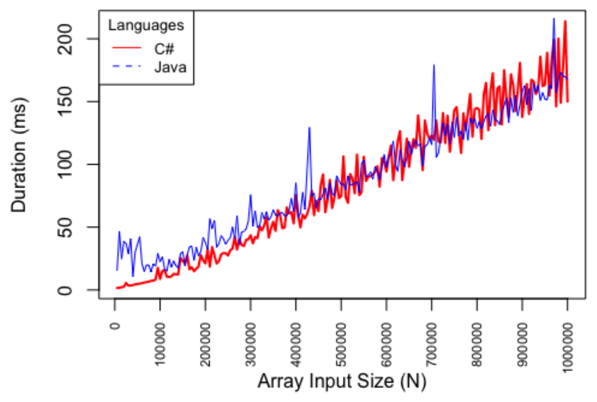

Fig. 8. C# vs. Java - 1024MB

As we gauge how the workload size in terms of array input size influences the languages' performance, Java and C# consistently outperform Python and Node.js across all memory sizes. According to Table 1, for 128MB, Java and C# had a median execution time of 1051.88ms and 703.68ms respectively. Comparatively, Python and Node.js obtain much larger durations of 8842.45 and 7584.17ms. C# performs the best at the lowest memory allocation. Furthermore, Java sees a performance increase when configured to 512MB and beats out C# on average. At the highest memory allocation of 1024MB, C# outperforms Java with the largest input size taking a duration of 213.91ms while achieving lower median and mean duration times compared to Java which took 216.44ms for the largest input as seen in Table 2. As the memory sizes increase, both Python and Node.js perform relatively better, however, Java and C# outperform again. Moreover, it is important to note that Python benefits more at the largest memory allocation of 1024MB in comparison to Node.js as they now both see similar performance in Figure 7 when before Node.js had a considerable advantage over Python under the other memory sizes as seen in Figures 5 and 6. Ultimately, it may be possible to categorise the languages into interpreted and compiled languages. Interpreted languages such as Python and Node.js encountered higher execution times than the compiled languages of Java and C#.

VI. Conclusion

The main objective of this thesis was to conduct a benchmark and evaluate the performance of the following languages on the AWS Lambda platform: C#, Java, Node.js, and Python. Additionally, tools to facilitate and ease testing and deployment were employed. The Serverless Framework aided development and deployment of functions. It allows for local test functions, and provides automated deployment to the chosen platform. Also, monitoring and logging tools such as Amazon's CloudWatch, and Artillery.io were applied to the data collection process. Initially, the research focused on how memory allocation affected the languages' performance, therefore an investigation was performed on three memory configurations: 128, 512, 1024MB. All languages saw increases in performance when allocated more than 128MB of memory. A move from 128 to 512MB saw improvements ranging from 70-79%. Moreover, a memory increase from 128 to 1024MB would result in performance increases ranging from 88-92%. The Java language benefited the most of all languages in this change which obtained a 92% reduction in execution time. A configuration from 512 to 1024MB saw diminished improvements where languages' execution time were reduced 53-60%. The second portion of the study revolved around how workload input size would impact the languages' performance. The end results presented that Java and C# invariably outperformed the other languages by a large extent with C# leading the benchmark at configurations of 128 and 1024MB, however, Java became the front runner at 512MB.

Future work of this thesis could include a repeat of the research methodology with the inclusion of the Go language, or other new language additions to the AWS Lambda platform.

Acknowledgement

We would like to thank our supervisor, Joel Scheuner, for his time and valuable feedback.

References

[1] Lynn, Rosati, Lejeune, and Emeakaroha, "A preliminary review of enterprise serverless cloud computing," *2017 IEEE 9th International Conference on Cloud Computing Technology and Science*, p162-163, 2017.

[2] ——, "A preliminary review of enterprise serverless cloud computing," *2017 IEEE 9th International Conference on Cloud Computing Technology and Science*, p162, 2017.

[3] S. B. May Al-Roomi, Shaikha Al-Ebrahim and I. Ahmad, "Cloud computing pricing models: A survey," *2013 International Journal of Grid and Distributed Computing*, p99, 2013.

[4] ——, "Cloud computing pricing models: A survey," *2013 International Journal of Grid and Distributed Computing*, p95, 2013.

[5] R. W. Wei-Tsung Lin, Chandra Krintz and M. Zhang, "Tracking causal order in aws lambda applications," *2018 IEEE International Conference on Cloud Engineering*, p50-51, 2018.

[6] "Cost optimization." [Online]. Available: https://aws.amazon.com/pricing/cost-optimization/ [Accessed: 24-Mar-2018].

[7] "What is cloud computing?" [Online]. Available: https://aws.amazon.com/what-is-cloud-computing/ [Accessed: 08-May-2018].

[8] "What is cloud computing?" [Online]. Available: https://cloud.google.com/what-is-cloud-computing/ [Accessed: 08-May-2018].

[9] "What is serverless computing?" [Online]. Available: https://azure.microsoft.com/en-us/overview/serverless-computing/ [Accessed: 08-May-2018].

[10] "Aws re:invent 2017: Getting started with serverless computing using aws lambda." [Online]. Available: https://www.slideshare.net/AmazonWebServices/getting-started-with-serverless-computing-using-aws-lambda-ent332-reinvent-2017 [Accessed: 08-May-2018].

[11] "Serverless framework documentation." [Online]. Available: https://serverless.com/framework/docs [Accessed: 24-Mar-2018].

[12] "Environment variables." [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/env_variables.html [Accessed: 08-May-2018].

[13] "Amazon ec2 instance types." [Online]. Available: https://aws.amazon.com/ec2/instance-types/ [Accessed: 08-May-2018].

[14] "Configuring lambda functions." [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html [Accessed: 08-May-2018].

[15] M. Fuller, *AWS Lambda: A Guide to Serverless Microservices*. Amazon Digital Services LLC, 2016.

[16] "Aws lambda documentation." [Online]. Available: https://aws.amazon.com/documentation/lambda [Accessed: 24-Mar-2018].

[17] “Aws lambda pricing.” [Online]. Available: https://aws.amazon.com/lambda/pricing [Accessed: 24-Mar-2018].

[18] Puripunpinyo and Samadzadeh, “Effect of optimizing java deployment artifacts on aws lambda,” *Computer Communications Workshops (INFOCOM WKSHPS)*, p443, 2017.

[19] Singh, Shukla, Chandra, and Dixit, “Performance evaluation of programming languages,” *Innovations in Information, Embedded and Communication Systems (ICIIECS)*, p978-p979, 2017.

[20] A. Hornsby and N. Undén, “Getting started with aws lambda and the serverless cloud,” *AWS Stockholm Summit*, 2016.

[21] “Comparing aws lambda performance when using node.js, java, `C#` or python.” [Online]. Available: https://read.acloud.guru/comparing-aws-lambda-performance-when-using-node-js-java-c-or-python-281bef2c7 [Accessed: 24-Mar-2018].

[22] “Artillery - a modern load testing toolkit.” [Online]. Available: https://artillery.io/ [Accessed: 24-Mar-2018].

[23] “Rosetta code.” [Online]. Available: http://www.rosettacode.org/wiki/Sort_an_integer_array [Accessed: 22-Apr-2018].

[24] “Rstudio.” [Online]. Available: https://www.rstudio.com/ [Accessed: 24-Mar-2018].

[25] “Aws lambda benchmark.” [Online]. Available: https://github.com/cocohub/aws-lambda-benchmarking [Accessed: 22-May-2018].