Using a Microbenchmark to Compare Function as a Service Solutions

Timon Back and Vasilios Andrikopoulos(✉)

University of Groningen, Groningen, The Netherlands
t.back@student.rug.nl, v.andrikopoulos@rug.nl

Abstract. The Function as a Service (FaaS) subtype of serverless computing provides the means for abstracting away from servers on which developed software is meant to be executed. It essentially offers an event-driven and scalable environment in which billing is based on the invocation of functions and not on the provisioning of resources. This makes it very attractive for many classes of applications with bursty workload. However, the terms under which FaaS services are structured and offered to consumers uses mechanisms like GB–seconds (that is, X GigaBytes of memory used for Y seconds of execution) that differ from the usual models for compute resources in cloud computing. Aiming to clarify these terms, in this work we develop a microbenchmark that we use to evaluate the performance and cost model of popular FaaS solutions using well known algorithmic tasks. The results of this process show a field still very much under development, and justify the need for further extensive benchmarking of these services.

Keywords: Function as a Service (FaaS) · Microbenchmark
Performance evaluation · Cost evaluation

1 Introduction

The wide adoption of cloud-native enabling technologies and architectural concepts like containers and microservices in the recent years has created an increasing interest in *serverless computing* as a programming model and architecture. In this model, code is executed in the cloud without any control of the resources on which the code runs [1]. Serverless encompasses a wide range of technologies, that following the discussion in [13] can be grouped into two areas: *Back-end as a Service (BaaS)* and *Function as a Service (FaaS)*. BaaS is especially relevant for mobile application development and is closely related to the SaaS delivery model, allowing the replacement of server-side components with third party services. Google's Firebase[1] is an example of such a service. FaaS, on the other hand is closer to the PaaS model, allowing individual business operations to be built and deployed on a FaaS platform. The key difference between FaaS and

[1] Firebase https://firebase.google.com/.

© IFIP International Federation for Information Processing 2018
Published by Springer Nature Switzerland AG 2018. All Rights Reserved
K. Kritikos et al. (Eds.): ESOCC 2018, LNCS 11116, pp. 146–160, 2018.
https://doi.org/10.1007/978-3-319-99819-0_11

PaaS is the scaling scope as discussed by Mike Roberts[2]: in PaaS the developer is still concerned with scaling an application up and down as a whole, while FaaS provides complete transparency to the scaling of functions, since this is handled by the platform itself.

There are a number of claimed benefits of serverless computing, and by extension also of FaaS, identified for example by [13]. More importantly, scaling becomes the responsibility of the platform provider and the application owner is charged only for how long a function is running as a response to its invocation (within a billable time unit—BTU). This is a big departure from the "traditional" model of cloud computing so far, at least when compared to other compute–oriented solutions like VM– and Container as a Service, where the owner is charged for provisioning these resources irrespective of their utilization. As a result, FaaS is perceived as the means to achieve significant cost savings, especially in the case of bursty, compute-intensive workloads [1] such as the ones generated by IoT applications.

At the same time, however, the pricing model of FaaS solutions can be difficult to decipher and surprisingly complex to model [2]. FaaS users are typically charged based on two components: *number of function invocations* across all functions belonging to the user, and *function execution duration* measured, confusingly enough, in *GB–seconds* per billing cycle. The first metric is relatively straightforward but potentially extremely dangerous in the case of decomposing application functionality into too many fine–grained functions that result into ever expanding cumulative costs. The second one is based on the practice of most FaaS providers, as discussed in the following section, of requiring the user to define a fixed memory amount to be allocated for each function execution. Users are then charged for the BTUs (in seconds) for which a function executed, multiplied by the allocated (or peak in the case of one provider) amount of memory in GB, times the per GB–seconds cost defined by the provider. FaaS adoption essentially also means loss of control over the performance of the functions themselves, since their execution is hidden under multiple layers of virtualization and abstraction by the platform providers, resulting into inconsistent performance results even for the same service and configuration [13].

With the aim of investigating and clarifying these two phenomena and their impact on FaaS adopters, this paper discusses the use of a *microbenchmark* in order to study how different FaaS solutions, and especially ones in the public cloud deployment model, behave in terms of performance and cost. More specifically, Sect. 2 presents the FaaS solutions that we will consider for the rest of this work and discusses related work. Section 3 incorporates a small set of algorithmic tasks with known computational and memory requirements in a microbenchmark of our design and implementation. Section 4 presents the results of executing the benchmark in a time window and discusses our findings while evaluating the selected FaaS solutions. Based on these findings we provide a series of lessons that we learned and that we believe are relevant for FaaS adopters in Sect. 5. Finally, Sect. 6 concludes this work with a short summary and future work.

[2] For more on the subject, see https://martinfowler.com/articles/serverless.html.

2 Background and Related Work

Since the introduction of Amazon Web Services Lambda[3] back in 2014 all major cloud providers have developed their own FaaS solution. Table 1 summarizes and compares the offerings of the most popular public Cloud providers [12]. More specifically, and in alphabetical order:

- *AWS Lambda* was the first FaaS public offering. At the time of writing, it offers memory usage to be specified in the $[128, 3008]$ MB interval in increments of 64 MB. It offers the most flexibility in terms of configuration options, and is the more mature of implementations from the offerings investigated by this work.
- *Google Cloud Functions*[4] is in beta status since its launch in February 2016. While the least flexible in terms of configuration options, Cloud Functions is the only of the FaaS solutions that clearly defines the amount of allocated CPU cycles per memory allocation option in its documentation.
- *IBM Cloud* (formerly known as IBM Bluemix) *Functions*[5] is based on the Apache OpenWhisk[6] FaaS platform implementation, allowing for easy hybrid deployment. It requires all functions to run as Docker containers, which allows for function development in any language.
- *Microsoft Azure Functions*[7], also launched in 2016, differs significantly from the other solutions in the sense that it does not expect the user to specify a fixed amount of memory to be used by the function in advance. The service bills only for the used memory per invocation, rounded up to the nearest 128 MB step, using at the same time the smallest billable time unit (1 ms).

In terms of related work, and considering how recently serverless computing was introduced, existing literature on the subject is relatively limited. Van Eyk et al. [3] for example identify the need for community consensus on what constitutes FaaS, and set the goal of developing an objective benchmark of FaaS platforms as a target for future work. The approaches presented by [8,15] investigate the cost of FaaS solutions as an infrastructural platform for the hosting of microservices. Their interest is in evaluating alternative deployment scenarios involving FaaS services and not with the performance of FaaS solutions themselves. The Costradamus approach [6] aims to measure the computation waste in FaaS usage accrued by monitoring function calls duration and contrasting them to billed BTUs. Both [5,14] use microbenchmarking of FaaS solutions in order to compare providers and calibrate their proposed systems, but for these works the comparison of providers is incidental and not the main focus. These works are therefore relevant but not directly related to the goals set for this work.

From more related works, [7,10] set out to explicitly benchmark and compare FaaS solutions in terms of performance and cost. While useful and insightful in

[3] AWS Lambda: https://aws.amazon.com/lambda/.
[4] Google Cloud Functions: https://cloud.google.com/functions/.
[5] IBM Cloud: https://console.bluemix.net/openwhisk/.
[6] Apache OpenWhisk: https://openwhisk.apache.org/.
[7] Microsoft Azure Functions: https://azure.microsoft.com/services/functions/.

Table 1. Comparison of the offerings by the major Cloud Service Providers (May 2018)

	Amazon WS Lambda	Google Cloud Functions	IBM Cloud Functions/Apache Open-Whisk	Microsoft Azure Functions
Memory Min	128 MB	128 MB	128 MB	128 MB
Memory Max	3008 MB	2048 MB	512 MB	1536 MB
Timeout Max	5 min	9 min	5 min	10 min
Billing Interval	100 ms	100 ms	100 ms	1 ms
Memory Allocation	Fixed	Fixed	Fixed	Dynamic
Natively Supported Languages	C# Go Java Node.js Python	Node.js	Java Node.js PHP Python Swift ...	C# F# Node.js
HTTP Invocation	✓	✓	✓	✓
HTTP plus Authentication	✓	—	✓	✓
Free Tier (One time/Periodical)	✓/✓	✓/✓	✓/✓	✓/✓

their own right, both works use much more coarse–grained tasks for their evaluation, focusing on concurrency and latency, respectively. The work by Malawski et al. [11] provides similar conclusions to ones discussed by this work, and in some ways supplements our findings with further insights; however it only discusses performance issues with FaaS solutions and does not investigate their impact on cost.

With this work, we focus on investigating the differences between the FaaS solutions presented above with respect to their compute/memory allocation policies, and their consequent effect on the cost model of cloud functions running on them.

3 Microbenchmark Design

As discussed in the previous section, and given the current lack of a FaaS benchmark, it becomes a common and necessary practice to use a microbenchmark for performance evaluation purposes. We chose a microbenchmark for this purpose since we aim to measure a basic feature of FaaS services (compute/memory allocation) for which a simple program should suffice, and because microbenchmarking is quite popular for cloud services evaluation [9]. The `faas-`μ`benchmark` is available online[8] and it actually contains more functions than the ones we explain in the following. In the interest of space, we limit the presentation of results to only three major functions from the microbenchmark.

[8] `faas-`μ`benchmark`: https://github.com/timonback/faas-mubenchmark.

Functions

The following functions were selected for inclusion in the `faas-`μ`benchmark` based on their characteristics with respect to their computational and memory requirements:

– Fast Fourier Transformation (FFT): performs an FFT computation using the Cooley-Tukey method as implemented by the `fft-js` library of Node.js (version 0.0.11)[9] for an increasing amount of discrete signals $k = 2^i, i \in \mathbb{N}^+$. The Cooley-Tukey method has computational complexity $O(NlogN)$ and is therefore representative of a moderate load to the system.
– Matrix Multiplication (MM): multiply square matrices of increasing size without any optimization (i.e. with complexity $O(n^3)$); the length of the matrices is defined as $n = i \times 100, i \in \mathbb{N}^+$, i.e. it increases by a step of 100 starting from 100.
– Sleep (S): sleep for $t = 2^i, i \in \mathbb{N}^+$ ms. This function is selected for evaluating the sensitivity of the FaaS offering to its invocation. Measured execution durations should in principle be equal to the specified parameter t, plus some initialization time.

Table 2 summarizes the characterization of the selected functions:

Table 2. Relative resource requirements for the benchmarking functions

Function	Computational	Memory
Fast Fourier Transformation (FFT)	Moderate	Moderate
Matrix Multiplication (MM)	High	High
Sleep (S)	Minimum	Minimum

The microbenchmark itself is highly configurable, allowing for subsetting or extending the parameter values for each function as desired by the user. All functions are implemented on top of the Node.js JavaScript runtime, since it is the execution environment that is common across all FaaS offerings (see Table 1).

Instrumentation

In order to reduce the complexity of the deployment process of the defined functions across different providers we decided to use the *Serverless framework*[10], as also adopted by [11]. This framework allows for the deployment of code to the majority of FaaS/serverless solutions by a simple command, assuming of course that an account has been created with the respective provider and the necessary authentication credentials have been provided to it. Since FaaS providers expect

[9] https://www.npmjs.com/package/fft-js.
[10] Serverless: https://serverless.com/.

different bindings for functions executed in their platform we created a custom minimal wrapper for each provider which reads the passed-in parameters, calls the appropriate function, and returns the result. The called algorithm is the same for every provider. The wrapper function is provided together with the rest of the microbenchmark as discussed above.

4 Services Evaluation

In the following we discuss how we use the `faas-`μ`benchmark` to compare the FaaS solutions presented in Sect. 2.

4.1 Evaluation Setup

Apache OpenWhisk is used as the baseline for the comparison between solutions. The February 2018 version from the OpenWhisk GitHub repository was deployed inside a VirtualBox machine (version 5.2.8) running Ubuntu Linux 14.04 LTS with 4 GB of memory allocated to it, on a notebook with a quad–core Intel i7–6700HQ (@2.6 GHz) and 8 GB of memory in total. The three functions discussed in the previous section (i.e. FFT, MM and S) are deployed on it, and on the FaaS solutions offered in the public cloud deployment model using the Serverless framework. Five configurations for each FaaS service are selected for comparison purposes by setting the allocated memory to $128, 256, 512, 1024$ and 2048 MB, and the functions are deployed in all of these configurations.

Looking at the comparison in Table 1, we need to clarify that IBM Cloud Functions/Apache OpenWhisk has a maximum allocation limit of 512 MB per function. However by building on Docker's memory management, more memory is addressable for function execution without terminating due to insufficient memory. As we will show in the following, this works quite well for most of the experiments we performed.

Moving on, in order to avoid potential differences among regions we try to keep the location of the deployments comparable (more specifically, AWS Lambda: `us-east-1`, Google Cloud Functions: `us-central-1`, Microsoft Azure Functions: `Central US`) with the exception of IBM Cloud Functions that were deployed in the United Kingdom region since this could not be changed for the free tier version that we are using for all experiments. The functions are invoked by a local machine at the University of Groningen using simply the curl command on the Linux OS; as we will discuss in the following, the location of the invoker does not affect any measurements, and it can therefore be placed anywhere it is deemed more convenient. Timeout is set for all solutions and configurations at 300 s (i.e. 5 min) except in the case of Google Cloud Functions where it is set to 540 s (9 min).

The microbenchmark was executed across 3 consecutive working days in the end of April 2018, resulting in three measurements per function and parameter for each service configuration. For each microbenchmark run we execute all three functions in Table 2 sequentially with their parameters ranging over the following intervals ($i \in \mathbb{N}^+$ in all cases):

1. S: $t = 2^i, i \in [1, 13]$
2. MM: $n = i \times 100, i \in [1, 10]$
3. FFT: $k = 2^i, i \in [13; 21]$

For each invocation we are measuring the execution duration as reported by the FaaS provider (i.e. without network latency affecting the measurements), the execution status (i.e. success or reported type of error), the billed duration, and the incurred cost for the function execution. All measurements are collected from the respective logs of each service and are aggregated as CSV files for each function for further processing. The measurements we report and analyze in the following are also available in the `faas-`μ`benchmark` repository under `/results/`.

4.2 Microbenchmark Results and Findings

Note: for the rest of this discussion we will be using the convention FunM, as a shorthand for function Fun $\in$ {FFT,MM,S} executed on a service configuration with M MBs of allocated memory, where M $\in \{128, 256, 512, 1024, 2048\}$, across all providers of interest. MM1024, for example, refers to the execution of the matrix multiplication function in configurations with 1024 MB of allocated memory in all providers, for all parameter values $n = [100, 1000]$ with step 100. For purposes of space saving, in the following we are also using only the provider's name instead of the full name of the FaaS solution, with the exception of Apache OpenWhisk which is simply shortened to OpenWhisk.

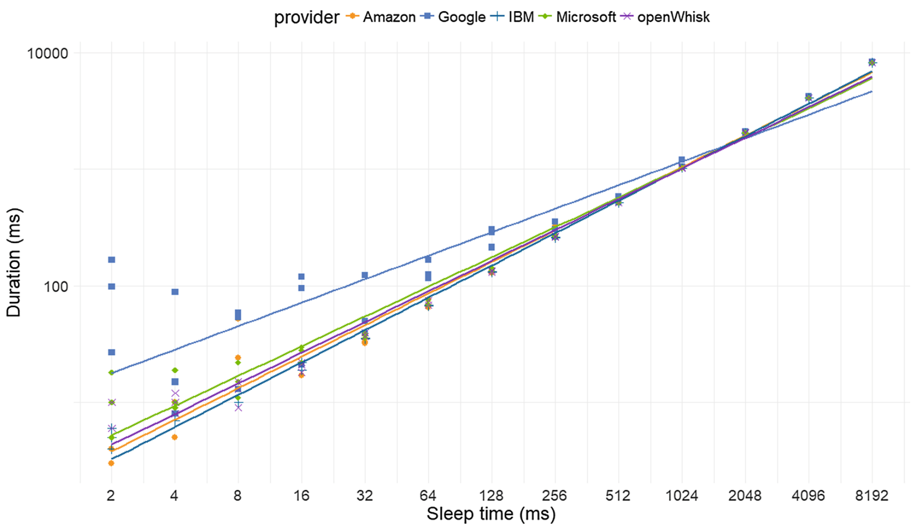

Fig. 1. Measured durations for S128 across all providers (log2–log plot). The straight lines show the fitted linear models to the observed data per provider.

Table 3. Mean Square Error (MSE) for linear regression to the observed data of S per provider for the different memory configurations.

Configuration	Provider				
	Amazon	Google	IBM	Microsoft	OpenWhisk
S128	265.82	2597.61	1.63	22.4	6.18
S256	62.46	1589.33	12.4	57.72	24.1
S512	41.96	726.93	1.79	20.04	12.06
S1024	31.62	757.52	2.03	14.63	15.96
S2048	12.31	851.3	2.4	18.75	5.72
mean (MSE)	81.03	1304.54	4.05	26.71	12.8

Sleep: With respect to function S, Fig. 1 shows the measured execution durations for S128. As it can be seen in the figure, the benchmarked FaaS solutions behave for the most part as expected, with a linear relation between execution time and sleep parameter t. This holds true however only after a sufficient large value of t—64 ms in our measurements—which is also around half of the BTU for all providers (except Microsoft, see Table 1). The solution that delays the most to converge into a linear relation with t, and at the same time exhibits the most variance, is actually the one by Google. This phenomenon appears also in the rest of the memory allocation configurations of this provider, as summarized by Table 3 which presents the mean square error (MSE) for the fitting of the measurements to a linear model with parameter t. The `lm` function of the R programming language (version 3.4.3) is used for the model fitting in Table 3. While the error in most configurations can be deemed acceptable, in the case of S128 as illustrated in Fig. 1 it is roughly ± 51 ms for the 128 MB configuration of Google Cloud Functions—that is, 50% of the service's BTU—and still an order or two magnitudes larger than the other ones in Table 3.

Matrix Multiplication: For MM we discuss our findings for the largest configurations (i.e. 1024 and 2048 MB), since we know that this function is the heaviest, at least in theory, of the functions that we include in the microbenchmark. Similar findings, but with the observed phenomena proportionally exaggerated are also concluded from the measurements in smaller configurations.

Figure 2 illustrates the collected measurements for progressively increasing matrix size n. Since we are in the normal–normal scale and we expect $O(n^3)$ complexity, we use the `loess` method of R for local polynomial regression fitting instead of the linear one. Looking at the measurements, it appears that the policy of Microsoft Azure Functions to assign memory dynamically instead of allocating it in advance is resulting in the relative worse among providers performance for this function as n grows. Further investigations in the effect of memory allocation in such calculations is necessary. On the other end of the spectrum, the OpenWhisk and consequently the IBM Cloud Functions solutions appear to be better able to handle the memory and computational requirements of this task

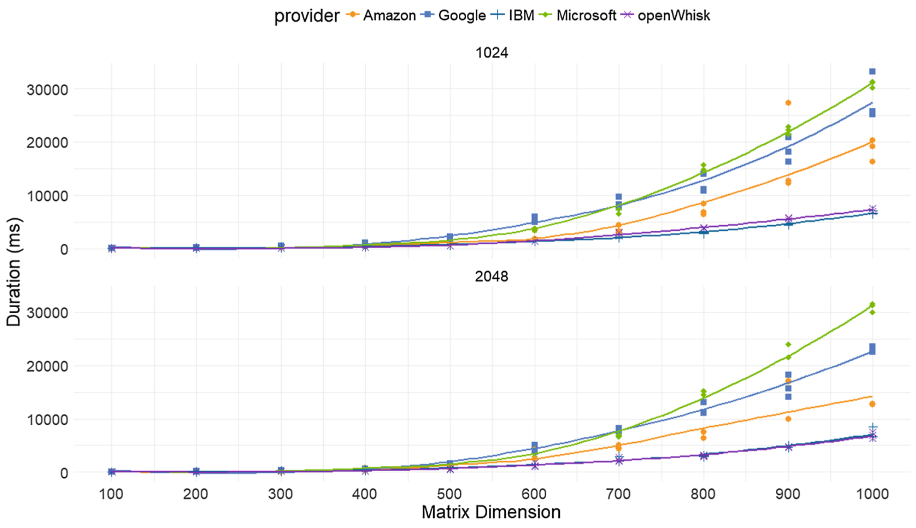

Fig. 2. Execution of MM1024 & MM2048 across all providers (norm–norm plots).

when compared to the other providers. It also seems that adding more memory to Amazon and Google's solutions results in better performance. Using only $n = 1000$ as a reference, the average execution times in these two solutions improve by 31.5% and 17.4%, respectively, when comparing the two configurations. We are going to use FFT to investigate this improvement in more depth in the following.

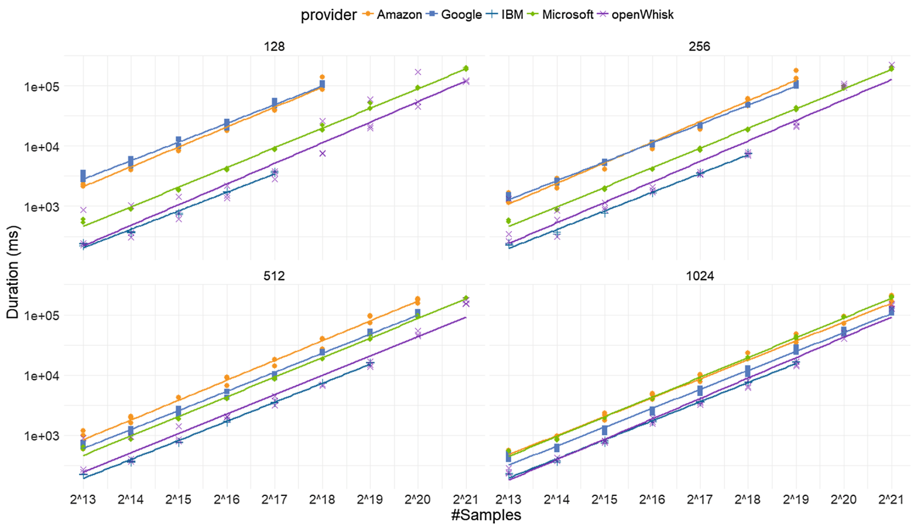

Fig. 3. Measured durations of successful executions of FFT128–FFT1024 across all providers (log2–log plots).

Table 4. Successful executions of FFT across all configurations per parameter k value.

k	Provider				
	Amazon	Google	IBM	Microsoft	OpenWhisk
[8192; 131072]	15	15	15	15	15
262144	15	15	12	15	15
524288	12	12	9	15	15
1048576	9	9	0	15	15
2097152	6	6	0	15	13
Total	∼86.7%	∼86.7%	∼71.1%	100%	∼98.5%

FFT: Figure 3 shows the reported execution durations of FFT across the first four memory configurations for comparison purposes, omitting any error responses. As it can be seen better in Table 4, only the dynamic memory allocation scheme of Microsoft Azure Functions allows for all values of parameter k to be calculated successfully. OpenWhisk is able to get additional memory from the local VM in order to calculate the FFT for k in most of the higher values, at the clear expense of speed however, as shown in Fig. 3. The figure also shows that for the rest of the providers, allocating more memory to the function results in more successful executions as k grows.

Zooming in on the interval of k values for which all FaaS solutions are able to successfully execute FFT, that is $k \in [8192; 131072]$ as shown in Table 4, we can study better the effect of memory allocation to the overall performance of each solution.

More specifically, as shown in Fig. 4, the solutions are separated into two groups. In the first group, the FaaS implementations by Microsoft and IBM/Apache do not meaningfully benefit from faster execution times by allocating more memory—in the former case because memory is actually allocated dynamically anyway, and in the latter because of the way OpenWhisk allows for partially dynamic memory allocation through its interaction with Docker. As shown in Table 4, however, the latter case can only cope with additional load so far before it starts producing error responses. In the second group, Amazon and Google's implementations clearly benefit from additional allocated memory, not only in terms of more successful executions, but also in terms of performance.

Focusing now on the cost incurred by the execution of FFT, Table 5 summarizes the cost calculation for all studied solutions[11] as *cumulative total (sum) cost* including all function invocations and consequent executions, and *mean cumulative cost* across configurations of 128 to 1024 MB per provider. While normalizing the cost per invocation may seem a more attractive option, the use

[11] OpenWhisk is deployed in a local VM, and therefore execution costs are not directly relevant; however for illustrative purposes we use the GB–seconds cost of IBM Cloud Functions for cost calculations. This makes the comparison between the private and public, in essence, deployment of OpenWhisk particularly interesting.

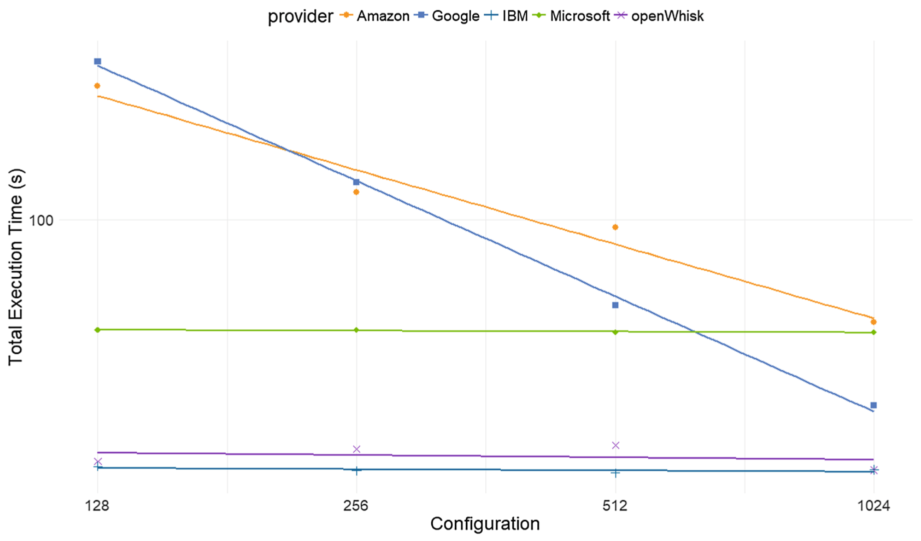

Fig. 4. Total duration per configuration and provider for FFT in seconds using only successful executions, i.e. $k \in [8192; 131072]$ (log2–log plot).

Table 5. Cumulative total and average costs per provider across all configurations for FFT in USD cents (April 2018 prices), respectively. See Footnote 11 for the cost calculation of OpenWhisk.

	Provider				
	Amazon	Google	IBM	Microsoft	OpenWhisk*
sum (*cost*)	2.832	1.941	0.258	3.305	2.228
mean (*cost*)	0.708	0.485	0.065	0.826	0.557

of cumulative costs fits better the interest of the consumer on the total cost of the FaaS service usage, especially given the observed variance we discussed in the previous.

As it can be seen from Table 5 and further reinforced by Fig. 5, when considering only successful function executions, IBM Cloud Functions is the most cost effective solution. Its high error rate due to its inability to deal with larger values of k has, however, to be taken seriously into consideration. Following on, Google's solution produces the next best solution in terms of cost, at the expense of high variability in its performance. Microsoft's solution on the other hand seems to be the most expensive and slow option, but at the time the one being able to scale better with k. Given the above, AWS Lambda seems to offer a good trade–off between performance, cost, and ability to cope with the requirements of the FFT function—but only if enough memory has been allocated per function.

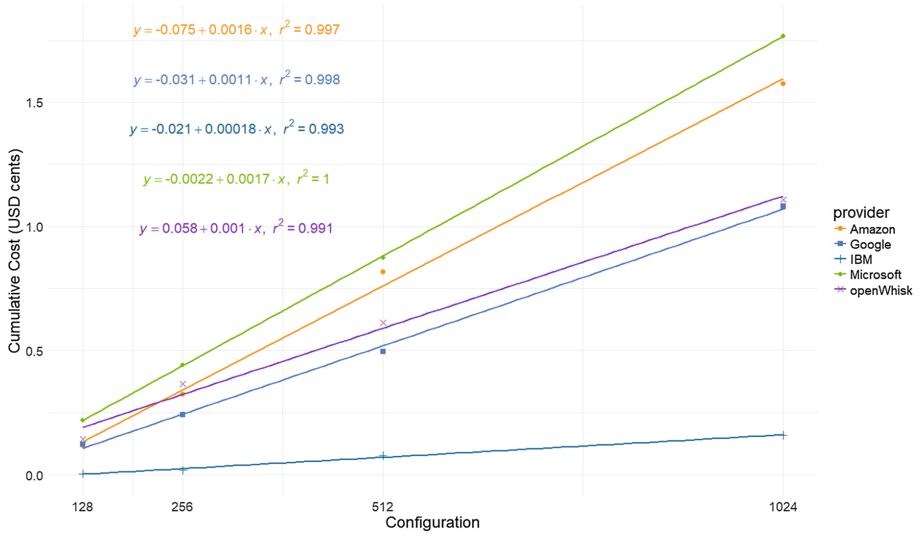

Fig. 5. Cumulative cost per provider and configuration for FFT in USD cents (April 2018 prices) with regression formulas (norm–norm scale).

5 Discussion and Lessons Learned

Before proceeding further, we have to identify the main threats to the validity of this work:

1. Not sufficient data points were collected during the microbenchmark execution to ensure the robustness of the findings. This is a known issue with this work and we plan to run it again for a longer period. Nevertheless, we can claim that anecdotally, the reported behavior of the FaaS solutions is consistent with any measurements we took outside of the reported ones in different days of April and May 2018. We are therefore confident in their validity, at least at this point in time.
2. Function implementation was done exclusively on Node.js; in principle, result replication is necessary in other programming languages but in the interest of time this is left as future work. In any case, as shown in Table 1, Node.js is the only common platform across all examined solutions. Comparing across programming languages could potentially only dilute the findings.
3. All measurements reported in the previous were taken on the free tier model offered by platform providers. We do not expect significant deviations when using the paid model, as the free tier seems to be a discount to have people try out (new) products. However, further experimentation is necessary in order to test this hypothesis.
4. The effect of the use of the Serverless framework for cross-provider deployment was not controlled; however we have no evidence of it affecting the validity of our measurements.

With respect to the lessons learned by the comparison of the various FaaS solutions, they can be summarized by the following:

1. The maturity of the examined FaaS solutions varies significantly when considering their observed performance. Especially Google's Cloud Functions seems to justify its label of beta state based on our measurements (see both Figs. 1 and 2).
2. There is a three–way trade–off between performance, cost, and ability to gracefully scale with each function's load before running out of memory or maximum execution time (see Figs. 2 and 3). Notice that there was no measurement with concurrent requests, so it is not possible to comment on the scaling of each solution with the overall load.
3. Adding more allocated memory only has a significant effect for some of the providers in terms of performance improvement (Fig. 4) and this has also been shown by [11]; however if the reliability of a function is important to the application developer then more memory is definitely recommended.
4. However, in addition to the above, it needs to be taken into account that while the relation between memory and cost appears to be linear, there is a significant difference between the coefficients of the cost functions per solution (see Fig. 5).
5. More extensive benchmarking of FaaS solutions is necessary in order to get a clearer picture of the state of play in FaaS solutions. As with the related works discussed in Sect. 2, this can extend beyond compute/memory evaluation to e.g. network and I/O parameters.

6 Conclusions and Future Work

In the previous sections we developed and used a microbenchmark in order to investigate two aspects of the Function as a Service (FaaS) sub–type of serverless computing: the differences in observable behavior with respect to the computer/memory relation of each FaaS implementation by the providers, and the complex pricing models currently being in use. For this purpose, we chose to include to our `faas-`μ`benchmark` three very common algorithmic tasks (Fast Fourier Transformation, matrix multiplication, and a simple sleep as a baseline), and implement them on top of the Node.js environment as the common denominator across the FaaS solutions under consideration. Executing the microbenchmark itself produced some unforeseen results with respect to the maturity of the offered solutions, and provided insights into the relation between performance and cost for software that is running in this cloud delivery model.

Future work is aimed at addressing the concerns discussed in the previous section. This entails proceeding with extensive benchmarking of the FaaS solutions across a longer period, considering also additional functions that impose different computational or memory constraints, and endeavor to clarify further the relation between memory and CPU cycle allocation. Potential differences between the perceived performance when functions are being executed in a free tier or not are also to be investigated. Furthermore, we also plan to expand the evaluation to OpenLambda [4], which is explicitly positioned as a research–oriented, non production–ready environment. The comparison with OpenWhisk as the only other open source solution would be particularly interesting. Finally, we aim to take the lessons learned by this work and put them into practice by developing instrumentation that allows application developers to route load across serverless or "traditional" IaaS resources in order to maximize their cost efficiency based on the characteristics of the application load.

References

1. Baldini, I., et al.: Serverless computing: current trends and open problems. In: Chaudhary, S., Somani, G., Buyya, R. (eds.) Research Advances in Cloud Computing, pp. 1–20. Springer, Singapore (2017). https://doi.org/10.1007/978-981-10-5026-8_1
2. Eivy, A.: Be wary of the economics of "serverless" cloud computing. IEEE Cloud Comput. **4**(2), 6–12 (2017)
3. van Eyk, E., Iosup, A., Seif, S., Thömmes, M.: The SPEC cloud group's research vision on FaaS and serverless architectures. In: Proceedings of the 2nd International Workshop on Serverless Computing, pp. 1–4. ACM (2017)
4. Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Serverless computation with OpenLambda. Elastic **60**, 80 (2016)
5. Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., Recht, B.: Occupy the cloud: distributed computing for the 99%. In: Proceedings of the 2017 Symposium on Cloud Computing, pp. 445–451. ACM (2017)
6. Kuhlenkamp, J., Klems, M.: Costradamus: a cost-tracing system for cloud-based software services. In: Maximilien, M., Vallecillo, A., Wang, J., Oriol, M. (eds.) ICSOC 2017. LNCS, vol. 10601, pp. 657–672. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69035-3_48
7. Lee, H., Satyam, K., Fox, G.: Evaluation of production serverless computing environments. Technical report, April 2018. https://doi.org/10.13140/RG.2.2.28642.84165
8. Leitner, P., Cito, J., Stckli, E.: Modelling and managing deployment costs of microservice-based cloud applications. In: Proceedings of IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC), pp. 165–174, December 2016
9. Li, Z., Zhang, H., O'Brien, L., Cai, R., Flint, S.: On evaluating commercial cloud services: a systematic review. J. Syst. Softw. **86**(9), 2371–2393 (2013)
10. Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., Pallickara, S.: Serverless computing: an investigation of factors influencing microservice performance. In: Proceedings of the IEEE International Conference on Cloud Engineering (IC2E 2018). IEEE (2018)
11. Malawski, M., Figiela, K., Gajek, A., Zima, A.: Benchmarking heterogeneous cloud functions. In: Heras, D.B., Bougé, L. (eds.) Euro-Par 2017. LNCS, vol. 10659, pp. 415–426. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75178-8_34
12. RightScale: RightScale 2018 State of the Cloud Report (2018). https://www.rightscale.com/lp/state-of-the-cloud

13. Roberts, M., Chapin, J.: What is Serverless? O'Reilly Media, Sebastopol (2017)
14. Spillner, J.: Exploiting the cloud control plane for fun and profit. arXiv preprint arXiv:1701.05945 (2017)
15. Villamizar, M., et al.: Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures. In: Proceedings of 16th IEEE/ACM International Symposium on Cluster Cloud and Grid Computing (CCGrid 2016), pp. 179–182, May 2016. https://doi.org/10.1109/CCGrid.2016.37