Received <day> <Month>, <year>; | Revised <day> <Month>, <year>; | Accepted <day> <Month>, <year>

DOI: xxx/xxxx

ARTICLE TYPE

# Performance Evaluation of Heterogeneous Cloud Functions

**Kamil Figiela | Adam Gajek | Adam Zima | Beata Obrok | Maciej Malawski***

AGH University of Science and Technology, Department of Computer Science, Krakow, Poland

**Correspondence**
*Maciej Malawski, Mickiewicza 30, 30-059 Krakow, Poland
Email: malawski@agh.edu.pl

## Abstract

Cloud Functions, often called Function-as-a-Service (FaaS), pioneered by AWS Lambda, are an increasingly popular method of running distributed applications. As in other cloud offerings, cloud functions are heterogeneous, due to variations in underlying hardware, runtime systems, as well as resource management and billing models. In this paper, we focus on performance evaluation of cloud functions, taking into account heterogeneity aspects. We developed a cloud function benchmarking framework, consisting of one suite based on Serverless Framework, and one based on HyperFlow. We deployed the CPU-intensive benchmarks: Mersenne Twister and Linpack, we measured the data transfer times between cloud functions and storage and we measured the lifetime of the runtime environment. We evaluated all the major cloud function providers: AWS Lambda, Azure Functions, Google Cloud Functions and IBM Cloud Functions. We made our results available online and continuously updated. We report on the results of the performance evaluation and we discuss the discovered insights on the resource allocation policies.

**KEYWORDS:**
Cloud computing, Serverless, FaaS, Cloud Functions, Performance Evaluation

## 1 | INTRODUCTION

Cloud Functions, pioneered by AWS Lambda, are becoming an increasingly popular method of running distributed applications. They form a new paradigm, often called Function-as-a-Service (FaaS) or serverless computing. Cloud functions allow the developers to deploy their code in the form of a function to the cloud provider, and the infrastructure is responsible for the execution, resource provisioning and automatic scaling of the runtime environment. Resource usage is usually metered with millisecond accuracy and the billing is per every 100 ms of CPU time used. Cloud functions are typically executed in a Node.js environment, but they also allow running custom binary code, which gives an opportunity for using them not only for Web or event-driven applications, but also for some compute-intensive tasks, as presented in our earlier work (1).

As in other cloud offerings, cloud functions are heterogeneous in nature, due to various underlying hardware, different underlying runtime systems, as well as resource management and billing models. For example, most providers use Linux as a hosting OS, but Azure functions run on Windows. This heterogeneity is in principle hidden from the developer by using the common Node.js environment, which is platform-independent, but again various providers have different versions of Node (as of May 2017: for AWS Lambda – Node 6.10, for Google Cloud Functions – Node 6.9.1, for IBM Cloud Functions – Node 6.9.1, for Azure Functions – 6.5.0.). Moreover, even though there is a common "function" abstraction for all the providers, there is no single standard API that would allow to switch providers easily.

In this paper, we focus on performance evaluation of cloud functions and we show how we faced various heterogeneity challenges. We have developed a framework for performance evaluation of cloud functions and applied it to all the major cloud function providers: AWS Lambda, Azure Functions, Google Cloud Functions and IBM Cloud Functions. Moreover, we used our existing scientific workflow engine HyperFlow (2) which has been recently extended to support cloud functions (1) to run parallel workflow benchmarks. We report on the initial results of the performance evaluation and we discuss the discovered insights on the resource allocation policies.

This paper is an extension of our earlier conference publication (3), in which we presented the preliminary results of our benchmarks. In this paper, we extend this publication by: (i) presenting the results of long running analysis of CPU performance for the period of April – September 2017 – March 2018, (ii) adding new measurements of data transfers and instance lifetime, and (iii) presenting the performance-cost analysis.

The paper is organized as follows. Section 2 discusses the related work on cloud benchmarking. In Section 3, we provide the motivation and formulate the research hypotheses. In Section 4, we outline our framework, while in Section 5, we give the details of the experiment setup. Section 6 presents the results, while their discussion is summarized in Section 7. Section 8 gives the conclusions and outlines the future work.

## 2 | RELATED WORK

Cloud performance evaluation, including heterogeneous infrastructures has been subject of previous research. Soon after first public cloud services were released, they received attention from the scientific community, resulting in benchmarks of Amazon EC2 from the perspective of scientific applications (4). More detailed studies of running scientific workflows in clouds, indicating performance and cost trade-offs were also conducted (5) using Pegasus Workflow Management system. An excellent example of cloud performance evaluation is in (6), where multiple clouds are compared from the perspective of many-task computing applications. Another set of extensive performance studies focusing on selected IaaS cloud providers were presented in (7), and impacts of virtualization and performance variability were emphasized. In addition to compute services, cloud storage has been also analyzed from the performance perspective (8). Several hypotheses regarding performance of public clouds are discussed in a comprehensive study presented in (9). All these studies provided not only up-to-date performance comparisons of current cloud provider offerings, but also reported on interesting insights regarding performance characteristics of clouds, effects of multitenancy, noisy neighbours, variability, and costs. No such comprehensive study of cloud functions has been performed so far.

Another group of of studies addressed not only standard IaaS platforms where virtual machines are provisioned on-demand, but also other options of cloud computing services. There are studies focused e.g. on burstable (10) instances, which are cheaper than regular instances but have varying performance and reliability characteristics. Performance of alternative cloud solutions such as Platform-as-a-Service (PaaS) has also been analyzed. E.g. (11, 12) focused on Google App Engine from the perspective of CPU-intensive scientific applications.

When serverless model and related infrastructures have emerged, pioneered by AWS Lambda, they have also become of interest to scientific community. A detailed performance and cost comparison of traditional clouds with microservices and the AWS Lambda serverless architecture is presented in (13), using an enterprise application. Similarly, in (14) the authors discuss the advantages of using cloud services and AWS Lambda for systems that require higher resilience. An interesting discussion of serverless paradigm is given in (15), where the case studies are blogging and media management application. Early experiments with scientific workflows using cloud functions (FaaS) infrastructures also report on performance evaluation. The first study was our earlier work, where we used Montage (16) workflow of small size to run it on on Google Cloud Functions and AWS Lambda. The second study used DEWE workflow engine to run also Montage using a hybrid setup combining IaaS and FaaS (17). Another example of price and performance comparison of cloud functions is provided also in (18), describing Snafu, a new implementation of FaaS model, which can be deployed in a Docker cluster on AWS. Its performance and cost is compared to AWS Lambda using a recursive Fibonacci cloud function benchmark. Recent works have also reported on the possibility to run Docker containers in AWS Lambda, with the performance studies focused on cold/hot start issues (19). None of the studies reported so far provides a comprehensive performance evaluation of all the major cloud function providers, giving the details on CPU performance, data access, overheads, lifetime and pricing, with emphasis on showing the heterogeneity of the environment.

We observe that the number of potential usage scenarios of these highly-elastic infrastructures is growing fast, and the interest of scientific community also increases (20, 21). Up to our knowledge, heterogeneous cloud functions have not been comprehensively studied yet, which motivates this research.

## 3 | MOTIVATION AND SCIENTIFIC QUESTIONS

Based on the related work, in order to explain the goals of our study and the selected benchmarks, we discuss the motivating use cases of cloud functions and scientific questions in the form of hypotheses.

### 3.1 | Motivating Use Cases

Serverless architectures, and cloud functions in particular, have numerous use cases in both commercial and scientific applications. These lead to usage patterns, which are common to cloud functions and thus are interesting from our performance evaluation perspective.

Typical scenarios are event-based applications, which need to react to events coming from Web or Mobile clients. These events trigger some actions in the system, which in turn are handled using cloud functions. For example, an upload of a image to the cloud storage may trigger a cloud function which will perform some image processing tasks. This results in a *download–process–upload* pattern, which is typical for cloud functions, since they are in principle stateless and cannot store intermediate data between the calls.

Similar scenarios can be observed in potential scientific use cases, where cloud functions can process compute-intensive tasks from e.g. scientific workflows (1, 17). Similar use cases can be seen in e.g. genomics data processing pipelines, where cloud functions (lambdas) can be run in parallel to work on individual data items from a large dataset (22). In this case also the *download–process–upload* pattern is prevalent.

Serverless infrastructures claim that they provide fast response time in reaction to events. There are, however, multiple software, hardware and networking layers in the whole execution environment, which inevitably introduce some overheads. It is thus interesting to measure how big these overheads are and whether the providers claims are supported by their implementations.

Cloud functions have a specific billing model, in which typically the cost is proportional to the execution time (with granularity of 100 milliseconds). The functions come also in various flavors (sizes) corresponding to the RAM allocated, and the CPU share is also proportional to the RAM. Therefore it is interesting to measure how exactly these performance guarantees stated in the documentation are actually implemented by the cloud providers and to which extent the users can rely on them.

## 3.2 | Scientific questions

The use cases, scenarios and patterns outlined above lead us to the following specific research questions, which we formulate below in the form of hypotheses we are going test using our benchmarking methodology.

Hypothesis 1: **Computational performance of a cloud function is proportional to function size.**

Documentation of cloud functions services states that CPU allocation is proportional to function size. Other resources are also supposed to be allocated proportionally.

Hypothesis 2: **Network performance (throughput) of a cloud function is proportional to function size.**

Other resources such as I/O and network are also expected to be allocated proportionally to the function size. Other than that, we do not expect differences in service latency.

Hypothesis 3: **Overheads do not depend on cloud function size and are consistent for each provider.**

We expect that the overheads are introduced mainly by the network connection and other components of the execution environment software stack, so there is no resource allocation policy involved which could influence them.

Hypothesis 4: **Application server instances are reused between calls and are recycled every couple of hours.**

It would cause significant latency if runtime environment was started separately for each request. We expect that execution environment is reused between requests. We also expect that even if runtime is reused, it will be recycled at most after a couple of hours. This is a common practice as there are possible memory leaks in the function code.

Hypothesis 5: **Functions are executed on heterogeneous hardware.**

Functions are running on top of IaaS infrastructure that do not guarantee exact hardware configuration. Usually, subscribers are only guaranteed CPU family, so they can make assumptions about supported instruction set. Differences in hardware specs may affect performance, and as a consequence billing.

To test these hypotheses we developed our benchmarking frameworks as described in Section 4 and we developed a set of benchmarks. The functionality of the benchmarks relevant to the hypotheses can be briefly summarized as follows.

- To evaluate hypothesis 1 we measure execution time of CPU-intensive workload.
- For hypothesis 2 we measure download and upload time of benchmark file.
- To test hypothesis 3 we compare request processing time that may be observed from the client with workload processing time measured in function runtime.

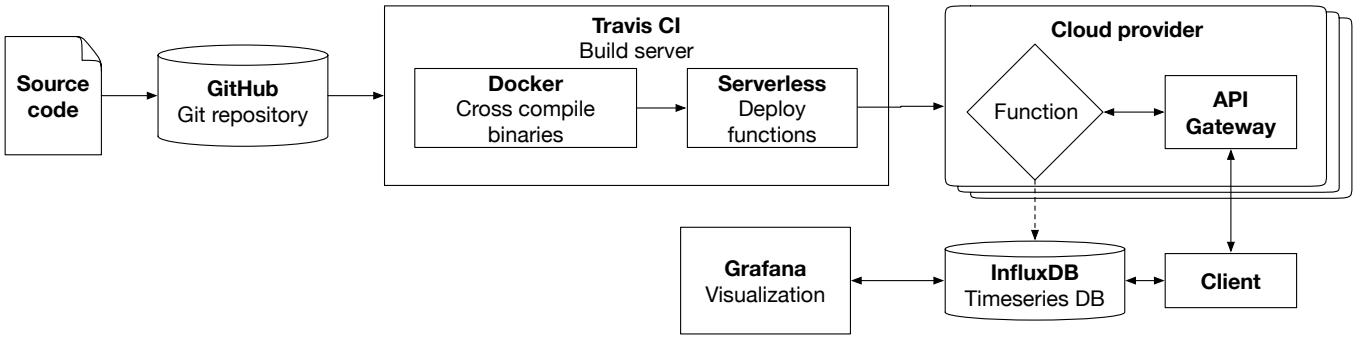


**FIGURE 1** Architecture of the cloud functions benchmarking framework based on Serverless Framework

- To test hypothesis 4, we assign an unique identifier for each execution environment and measure for how long it can be observed.
- To evaluate hypothesis 5 we determine processor type used for each function call if possible.

These hypotheses and their evaluation methods lead us to the development of benchmarking frameworks and to design of the specific experiments, which we describe in the following sections.

## 4 | BENCHMARKING FRAMEWORK FOR CLOUD FUNCTIONS

For benchmarking cloud function providers, we used two frameworks. The first one is our new suite, designed specifically for this research, based on Serverless Framework. The second one uses our HyperFlow workflow engine (2, 1). The reason for having two suites was that one of them, namely the HyperFlow, has been already used before to run preliminary experiments on cloud functions, and it allows us to execute workflows which can have many parallel tasks. Therefore it was natural to use as a driver for our CPU-intensive benchmarks. On the other hand, we needed a new suite, which would handle heterogeneity of the platforms, and at the same time allow running the tasks over a long period of time, continuously monitoring the performance of the infrastructure. We plan to integrate these two suites in the future.

### 4.1 | Suite Based on Serverless Framework

The objective of this benchmarking suite is to execute and gather performance results of heterogeneous cloud function benchmarks over a long period of time. The suite has to run as a permanent service and execute selected benchmarks periodically. The results are then stored and available for examination. Our goal was to automate functions deployment as much as possible to improve results reproducibility. The architecture of the suite is shown in Fig. 1 .

In order to deploy our benchmark suite we have used the Serverless Framework[1]. It provides a uniform way of setting up cloud functions deployment and supports, at the time of writing, AWS Lambda, IBM Cloud Functions and Azure Functions natively and Google Cloud Functions through an official plugin. In order to streamline our data taking process, we automated code deployment even further by setting up project on Travis continuous integration (CI), so that the code is automatically deployed on each cloud whenever we push new code to the Git repository. This also simplified security credentials management, since we do not need to distribute deployment credentials for each provider.

To address the heterogeneity of runtime environments underlying cloud functions, we have created dedicated wrappers for native binary that was executed by the function. We have used Docker to build binaries compatible with target environments. For Linux based environments, we use amazonlinux image to build a static binary that is compatible with AWS Lambda, Google Cloud Functions and IBM Cloud Functions. Azure Functions run in a Windows-based environment, thus it requires a separate binary. We used Dockcross[2] project that provides a suite of Docker images with cross-compilers, which includes a Windows target.

The Serverless Framework is able to deploy functions with all the necessary companion services (e.g. HTTP endpoint). However, we still had to adapt our code slightly for each provider, since the required API is different. For instance, AWS Lambda requires a callback when a function result is ready, while IBM Cloud Functions requires to return a Promise for asynchronous functions. The cloud platforms also differ in how $PATH environment variable and current working directory are handled.

[1]https://serverless.com
[2]https://github.com/dockcross/dockcross

The benchmarks results are sent to the InfluxDB time series database. We have also setup Grafana for convenient access to benchmark results. We implemented our suite in Elixir and Node.js. The source code is available on GitHub[3].

### 4.2 | Suite Based on HyperFlow

For running parallel benchmarking experiments we adapted HyperFlow (2) workflow engine. HyperFlow was earlier integrated with Google Cloud Functions (1), and for this work it was extended to support AWS Lambda. HyperFlow is a lightweight workflow engine based on Node.js and it can orchestrate complex large-scale scientific workflows, including directed acyclic graphs (DAG).

For the purpose of running the benchmarks, we used a set of pre-generated DAGs of the fork-join pattern: the first task is a fork task which does not perform any job, it is followed by $N$ identical parallel children of benchmark tasks running the actual computation, which in turn are followed by a single join task which plays the role of a final synchronization barrier. Such graphs are typical for scientific workflows, which often include such parallel stages (bag of tasks), and moreover are convenient for execution of multiple benchmark runs in parallel.

In the case of HyperFlow, the cloud function running on the provider side is a JavaScript wrapper (HyperFlow executor), which runs the actual benchmark, measures the time and sends the results to the cloud storage, such as AWS S3 or Google Cloud Storage, depending on the cloud provider.

## 5 | EXPERIMENT SETUP

We configured our frameworks with two types of CPU-intensive benchmarks, one focused on integer and the other on floating-point performance.

### 5.1 | Configuration of the Serverless Benchmarking Suite

The configuration of the benchmark suite consists of the following parts: (1) Integer-based CPU intensive benchmark, (2) Instance lifetime, (3) Data transfer benchmark.

Cloud services may be deployed in multiple geographical locations called regions. For all these parts, on AWS Lambda functions were deployed in eu-west-1 region, on Google Cloud Functions functions were deployed in us-central1 region, on IBM Cloud Functions functions were deployed in US South region and on Azure Functions function was deployed in US West region. Such setup results from the fact that not all of the providers offer cloud functions in all their regions yet.

#### Integer-based CPU intensive benchmark

In this experiment we used a random number generator, as an example of an integer-based CPU-intensive benchmark. Such generators are key in many scientific applications, such as Monte Carlo methods, which are good potential candidates for running as cloud functions.

Specifically, the cloud function is a JavaScript wrapper around the binary benchmark, which is a program written in C. We used a popular Mersenne Twister (MT19937) random number generator algorithm. The benchmark runs approximately 16.7 million iterations of the algorithm using a fixed seed number during each run and provides reproducible load. The benchmark executes in 5 seconds on modern laptop.

We measure the execution time $t_b$ of the binary benchmark from within the JavaScript wrapper that is running on serverless infrastructure, and the total request processing time $t_r$ on the client side. We decided to deploy our client outside the clouds that were subject to examination. The client was deployed on a machine hosted in Scaleway cloud in Paris datacenter. The benchmark was executed for each provider every 5 minutes. We took multiple measurements for different memory sizes available: for AWS Lambda – 128, 256, 512, 1024, 1536 MB, for Google Cloud Functions – 128, 256, 512, 1024, 2048 MB, for IBM Cloud Functions – 128, 256, 512 MB. Azure Functions do not provide a choice on function size and the memory is allocated dynamically. The measurements: binary execution time $t_b$ and request processing time $t_r$ were sent to InfluxDB by the client. Since the API Gateway used in conjunction with AWS Lambda restricts request processing time to 30 seconds and function performance is proportional to memory allocation, we were not able to measure $t_r$ for 128 MB Lambdas. Although the requests timeout on the API Gateway, the function completes execution. In this case, the function reports $t_b$ time directly to InfluxDB.

We started collecting data on April 18, 2017, and the data used in this paper include the values collected till Sep 21, 2017.

[3]https://github.com/kfigiela/cloud-functions

### Instance lifetime

We observed that providers reuse the same execution environment, i.e. Node.js process, to process subsequent requests. In order to investigate this sound approach, we measure $t_l$ which is maximum recorded lifetime of each execution environment process. To do this we assign a global variable with timer value when the execution environment is started. Then, we return the time elapsed since with response to every request. To distinguish one execution environment from another we either use MAC address of virtualized network adapter (AWS) or use random identifier that we assign to another global variable (other providers).

For this benchmark, we present data from September 19, 2017 till March 10, 2018.

### Data transfer benchmark

We have also deployed second set of functions for AWS Lambda and Google Cloud Functions where we replaced our CPU-intensive benchmark with a procedure that measures time required to download and upload 64 MB file from object storage. We chose this file size so that the transfer time is between 1 second and 30 seconds, where we can expect that the transfer rate dominates the latency. As object storage we used Amazon S3 for AWS Lambda and Google Cloud Storage for Google Cloud Functions. Those measurements are also reported to Influx.

For this benchmark, we present data from October 23, 2017 till March 10, 2018.

## 5.2 | Configuration of HyperFlow Suite

As a benchmark we used the HPL Linpack[4], which is probably the most popular CPU-intensive benchmark focusing on the floating point performance. It solves a dense linear system of equations in double precision and returns the results in GFlops. To deploy the Linpack on multiple cloud functions, we used the binary distribution from Intel MKL[5], version mklb_p_2017.3, which has binaries for Linux and Windows.

As benchmark workflows we generated a set of fork-join DAGs, with parallelism $N = [10, 20, ..., 100]$ and $N = [200, 400, 800]$, thus it allowed us to run up to 800 Linpack tasks in parallel. Please note that in this setup all the Linpack benchmarks run independently, since cloud functions cannot communicate with each other, so this configuration differs from the typical Linpack runs in HPC centers which use MPI. Our goal is to measure the performance of individual cloud functions and the potential overheads interference between parallel executions. The Directed Acyclic Graph (DAG) representing our Linpack workflow is shown in Fig. 2

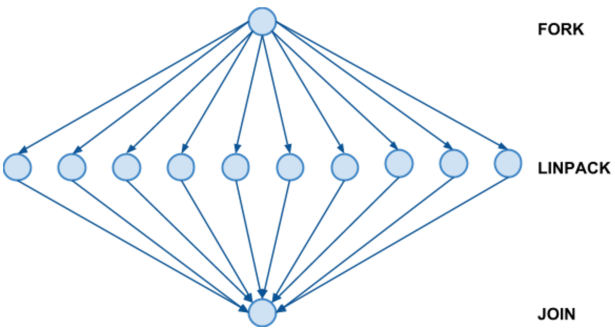


**FIGURE 2** Example DAG for $N = 10$ representing our parallel execution of Linpack benchmark.

The Linpack was configured to run using the problem size (number of equations) of $s \in \{1000, 1500, 2000, 3000, 4000, 5000, 6000, 8000, 10000, 12000, 15000\}$ Not all of these sizes are possible to run on functions with smaller memory, e.g. $4000 \times 4000 \times 8B = 128MB$, so the benchmark stops when it cannot allocate enough memory, reporting the best performance $p_f$ achieved (in GFlops).

We run the Linpack workflows for each $N$ on all the possible memory sizes available on Google Cloud Functions (128, 256, 512, 1024, 2048 MB) and on AWS Lambda on sizes from 128 to 1536 with increments of 64 MB.

On AWS Lambda functions were deployed in eu-west-1 region, on Google Cloud Functions functions were deployed in us-central1 region.

[4] http://www.netlib.org/benchmark/hpl/
[5] https://software.intel.com/en-us/articles/intel-mkl-benchmarks-suite

## 6 | PERFORMANCE EVALUATION RESULTS

Our benchmarks from the serverless suite run permanently and the original unfiltered data as well as current values are available publicly on our website[6]. They include also selected summary statistics and basic histograms. The data can be exported in CSV format, and we included the data in the GitHub repository.

Selected CPU performance results are presented in the following subsections. First we report on integer (6.1) and floating point (6.2) operations performance. Then, we present data transfer times (6.3) and overheads (6.4). Finally, we measure the lifetime of instances (6.5) and analyze the costs (6.6). The results are accompanied by their discussion and important observations.

### 6.1 | Integer Performance Evaluation

The results of the integer benchmarks using Mersenne Twister random generator are presented in Fig. 3 . They are shown as histograms, grouped by providers and function size. They give us interesting observations about the resource allocation policies of cloud providers.

Firstly, the performance of AWS Lambda is fairly consistent, and agrees with the documentation which states that the CPU allocation is proportional to the function size (memory).

On the other hand, Google Cloud Functions execution time have bi-modal distributions with higher dispersion. All the functions with memory smaller than 2048 MB have two peaks: one around the expected higher values (depending on the memory allocated), and the second peak overlapping with the performance of the fastest 2048 MB function. This suggests that Google Cloud Functions does not enforce strictly the performance limits, and opportunistically invokes smaller functions using the faster resources.

To better show this distribution of Google Cloud Functions execution times, we plot them using the logarithmic scale in Fig. 4 . The peaks on the left are clearly visible. The cases, when the function executes faster than expected are relatively rare: by counting the number of the events when the execution time is smaller than 10 seconds, we estimate that it occurs in less than 5% cases.

Regarding IBM Cloud Functions, the performance does not depend on the function size, and the distribution is quite narrow, as in the case of AWS Lambda.

On the other hand, the performance of Azure has much wider distribution, and the average execution times are relatively slower. This can be attributed to different hardware, but also to the underlying operating system (Windows) and virtualization technology.

The variability of the results is highly spread among the providers. AWS Lambda is the most consistent in performance: the standard deviation of the sampled data is 1.6 s for 128 MB, 0.4 s for 256 MB and only 0.12 s for the 1536 s. Google Cloud Functions has the standard deviation of 8 s for 128 MB, 5 s for 256, and 0.5 s for 2048. The fastest function is also the most consistent in performance, also due to the the lack of multiple peaks in the distribution. For IBM Cloud Functions, all the function sizes have the standard deviation of about 1 second, which is a results of some outliers. Finally, Azure Functions has the standard deviation of nearly 3 seconds, which is visible in a wide distribution and a long tail of outliers.

We provide the number of entries for each histogram. These numbers are not equal for all providers and function sizes, since some of the data points from our long-running experiments are missing. This may be due to transient errors or timeouts that cloud functions infrequently exhibit, and some of them may be caused by the failures on our client side. These failures were not frequent, so we consider them as negligible taking into account the time-span of the experiment.

### 6.2 | Floating-point Performance Evaluation

Results of the Linpack runs are shown in Fig. 5 , as scatter-plots where density of circles represents the number of data points. AWS Lambda data consists of over 12,000 points, and Google Cloud Functions of over 2,600 points. We show also histograms of subsets of these data, selected for brevity.

In the case of AWS, we observe that the maximum performance grows linearly with the function size. There is, however, a significant portion of tasks that achieved lower performance. With the growing memory, we can see that the execution times form two clusters, one growing linearly over 30 GFlops, and one saturating around 20 GFlops.

In the case of Google, we observe that the performance of tasks is clustered differently. The performance of one group of tasks grows linearly with memory. On the other hand, there is a large group of tasks, which achieve the top performance of 15 GFlops regardless of the function size. Interestingly, we observed that the smallest functions of 128MB always achieved the best performance of about 14 GFlops.

[6] http://cloud-functions.icsr.agh.edu.pl

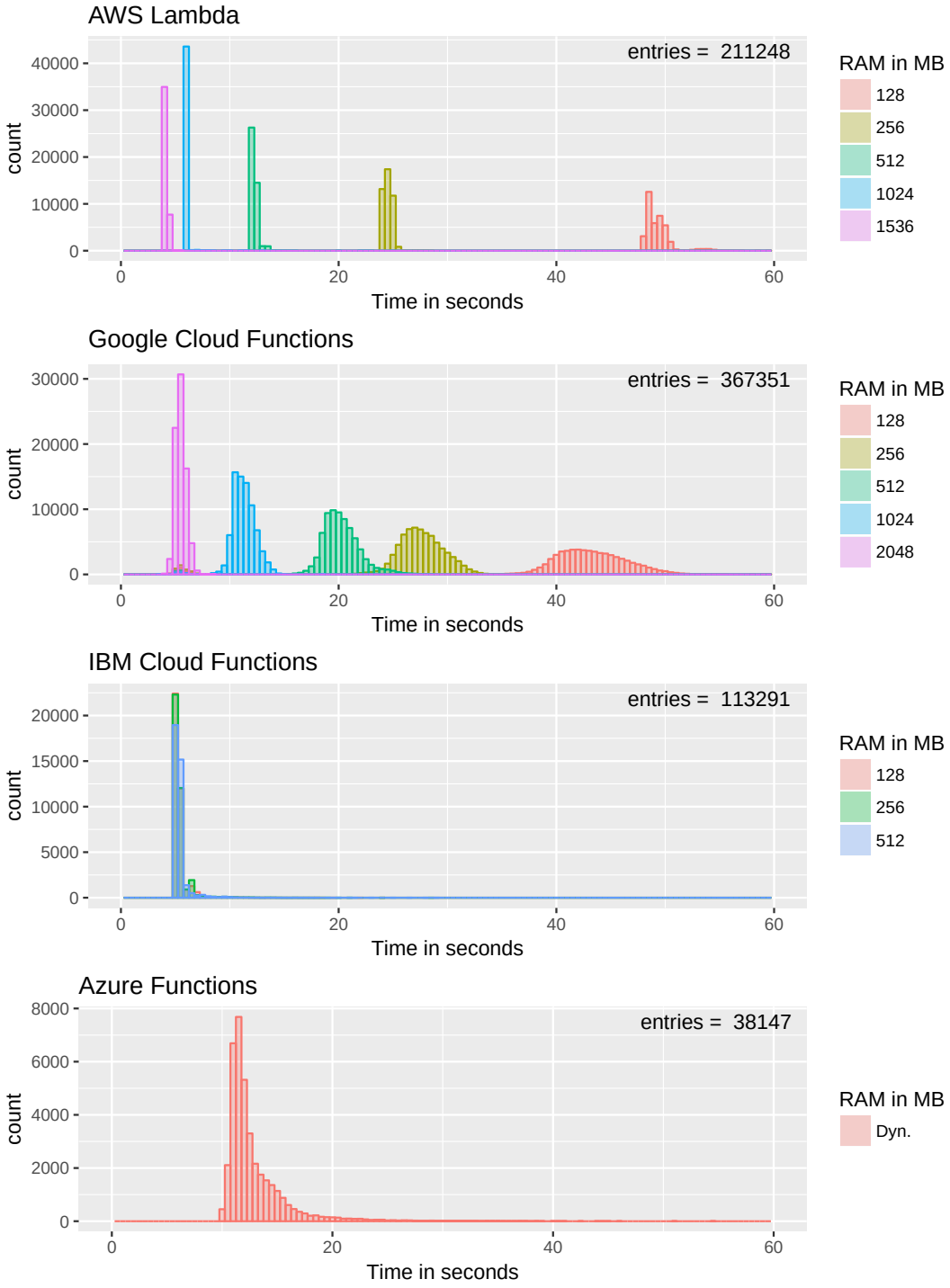


**FIGURE 3** Histograms of integer-based MT random number generator benchmark execution time vs. cloud function size. In the case of Azure Functions memory is allocated dynamically.

To illustrate the multimodal nature of performance distribution curves of Google Cloud Functions, we show the results as histograms in Fig. 5 for selected memory sizes. As in the case of integer-based performance tests, the AWS Lambda show much more consistent results, while for Google Cloud Functions the performance points are clustered.

The most interesting observation is regarding the scheduling policies of cloud providers, as observed in both MT and Linpack experiments. Both Google and AWS claim that the CPU share for cloud functions is proportional to the memory allocated. In the case of AWS we observe a fairly

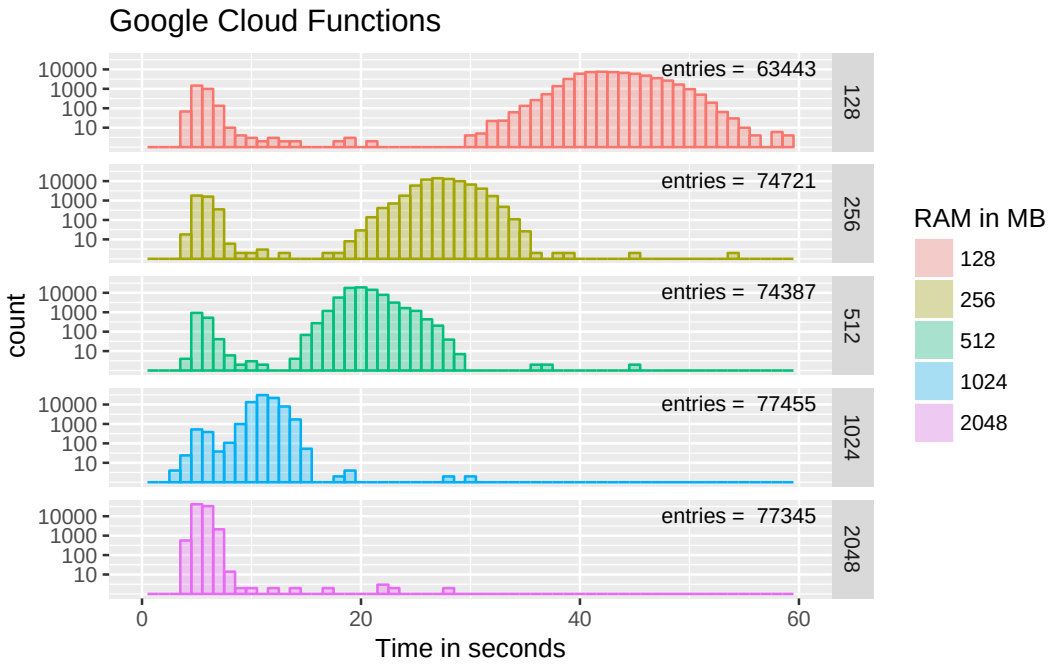


**FIGURE 4** More detailed histogram of Google Cloud Functions results for integer-based benchmark shown in logarithmic scale.

linear performance growth with the memory size, both for the lower bound and the upper bound of the plot in Fig. 5 . In the case of Google, we observe that the lower bound grows linearly, while the upper bound is almost constant. This means that Google infrastructure often allocates more resources than the required minimum. This means that their policy allows smaller functions (in terms of RAM) to run on faster resources. This behavior is likely caused by optimization of resource usage via reuse of already spawned faster instances, which is more economical that spinning up new smaller instances. Interestingly, for Azure and IBM we have not observed any correlation between the function size and performance.

Another observation is the relative performance of cloud function providers. AWS achieves higher scores in Linpack (over 30 GFlops) whereas Google tops at 17 GFlops. Interestingly, from the Linpack execution logs we observed that the CPU frequency at AWS is 3.2 GHz, which suggests Xeon E5-2670 (Ivy Bridge) family of processors, while at Google Cloud Functions it is 2.1 GHz which means Intel Xeon E5 v4 (Broadwell). Such difference in hardware definitely influences the performance. These Linpack results are confirmed by the MT benchmark. Since we have not run Linpack on Azure and IBM yet, we cannot report on their floating point performance, but the MT results also suggest the differences in hardware.

Summing up, the results of both integer and floating-point performance benchmarks confirm that hypothesis 1 is true for AWS Lambda and Google Cloud Functions, with the exception for the about 5% cases when Google allows the smaller functions to run faster. On the contrary, IBM and Azure do not conform to this rule.

## 6.3 | File Transfer Times to/from Cloud Storage

Fig. 6 and 7 show distribution of download and upload times for cloud object storage. We may observe that transfer times are proportional to memory allocation. Similarly to CPU benchmark we observe bi-modal distribution for small functions hosted on Google.

The measurements of file transfer times between cloud functions and object storage shows clearly that both the download and upload times depend on the size of the function. AWS exhibits much shorter data transfer times than Google cloud, and also the smaller variance. In Google we observe similar bi-modal distributions in Fig. 6 and 7 , which confirms our observation that Google often schedules smaller functions on faster instances, which is observed also for CPU-intensive benchmarks. We thus confirmed that our hypothesis 2 is true, with the exceptions resulting from the Google Cloud Functions resource allocation policy.

## 6.4 | Overheads Evaluation

By measuring the binary execution time $t_b$ inside the functions as well as the request processing time $t_r$ (as seen from the client), we can also obtain a rough estimate on total overhead $t_o = t_r - t_b$. The overhead includes: network latency, platform routing and scheduling overheads. Endpoints

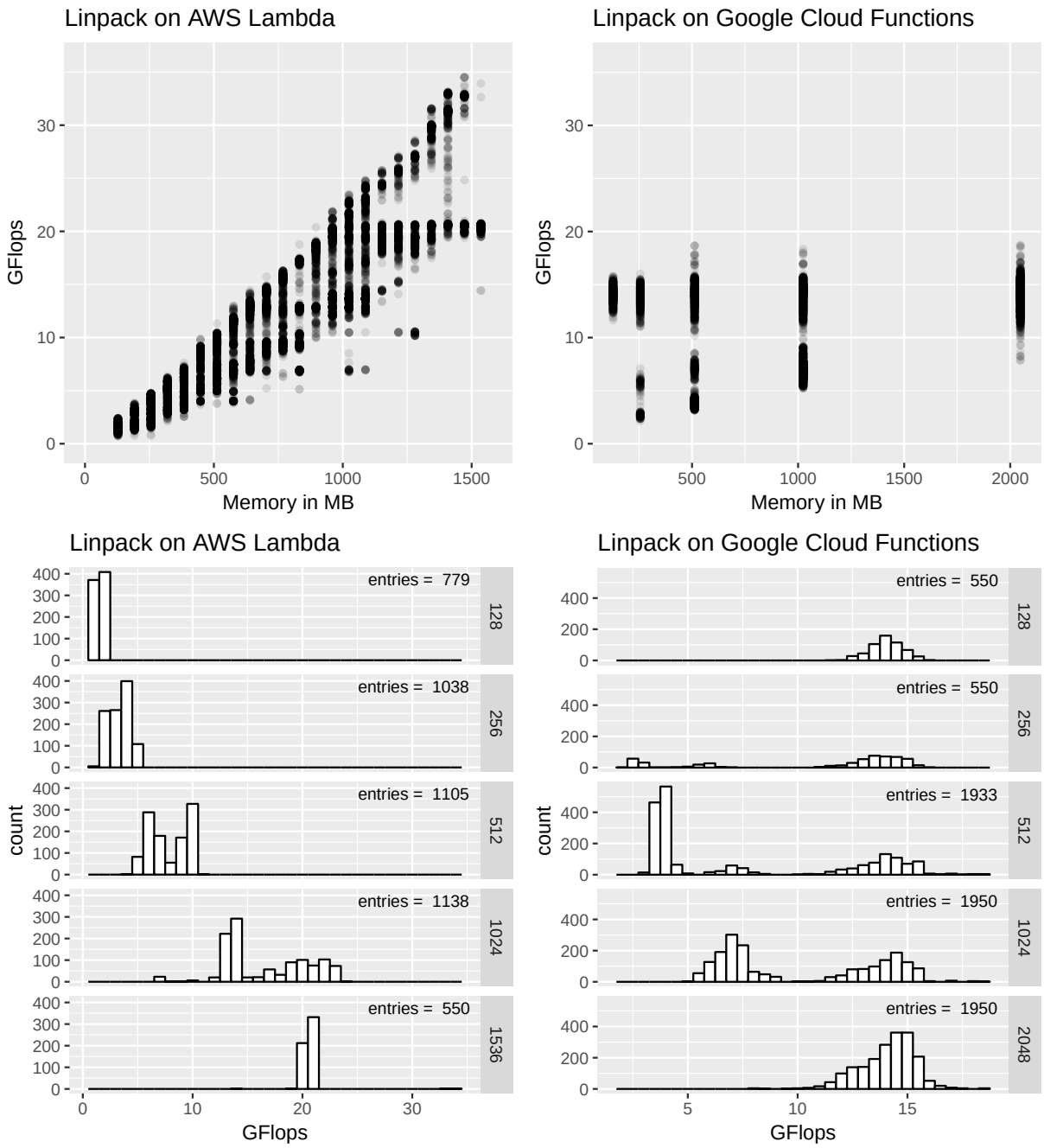


**FIGURE 5** Linpack performance versus cloud function size. The results were collected using the suite based on HyperFlow.

exposed by cloud providers are secured with HTTPS protocol. We warmed up the connection before performing each measurement, so that we were able to exclude the TLS handshake from $t_r$. Unfortunately, we could not measure the network latency to the clouds as AWS and Google provide access to functions via CDN infrastructure. The average round trip latency (ping) to IBM Cloud Functions was 117 ms and 155 ms to Azure Functions.

Histograms of $t_o$ are presented in Fig. 8 . We may see that the latency is lowest for AWS Lambda as both benchmark function and client server were located in Europe, for other providers our requests had to be routed overseas. Nevertheless, one may observe that overhead is stable with a few outliers. However, for IBM one may see that there are two peaks in the distribution.

Furthermore, we measured $t_r$ for requests targeting an invalid endpoint. This gives a hint on network latency under the assumption that invalid requests are terminated in an efficient way. The average results were consistent with typical network latency: for AWS Lambda – 43 ms, for Google

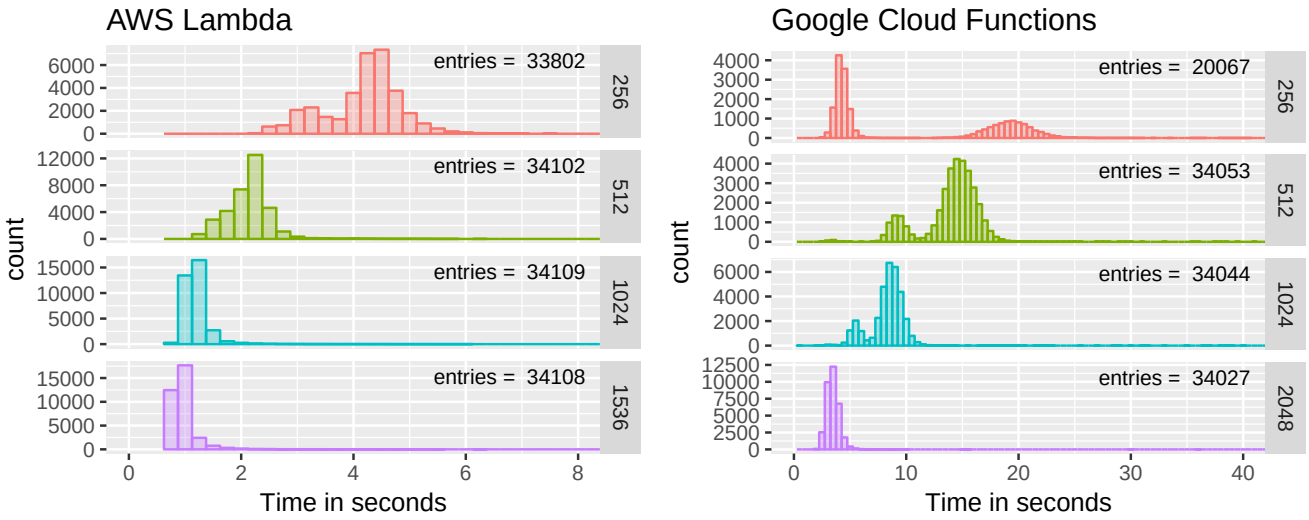


**FIGURE 6** Distribution of download times for 64 MB files for cloud function providers AWS and Google.


**FIGURE 7** Distribution of upload times for 64 MB files for cloud function providers AWS and Google.

Cloud Functions – 150 ms, for IBM Cloud Functions – 130 ms. However, for Azure the latency measured that way was 439 ms which is significantly larger than the network ping time. We did not observe any correlation between the function size and the overheads. We thus consider hypothesis 3 as valid.

## 6.5 | Instance Lifetime

Node.js process lifetime histograms for data gathered between September 19, 2017 and March 10, 2018 are shown in Fig. 9 . One may notice that distributions variy between providers. On Azure, the environment process is being preserved of a very long time up to two weeks. This confirms Hypothesis 4 we stated in Section 3.2. On AWS Lambda the Node.js environment is recycled every a few (up to 8) hours. IBM Cloud Functions recycles execution environment within a few hours, however we observed some long-living processes. Interestingly, we observe that on Google Cloud Functions, environments with low memory allocation are terminated more frequently, while the longer lifetime is being observed for larger allocations.

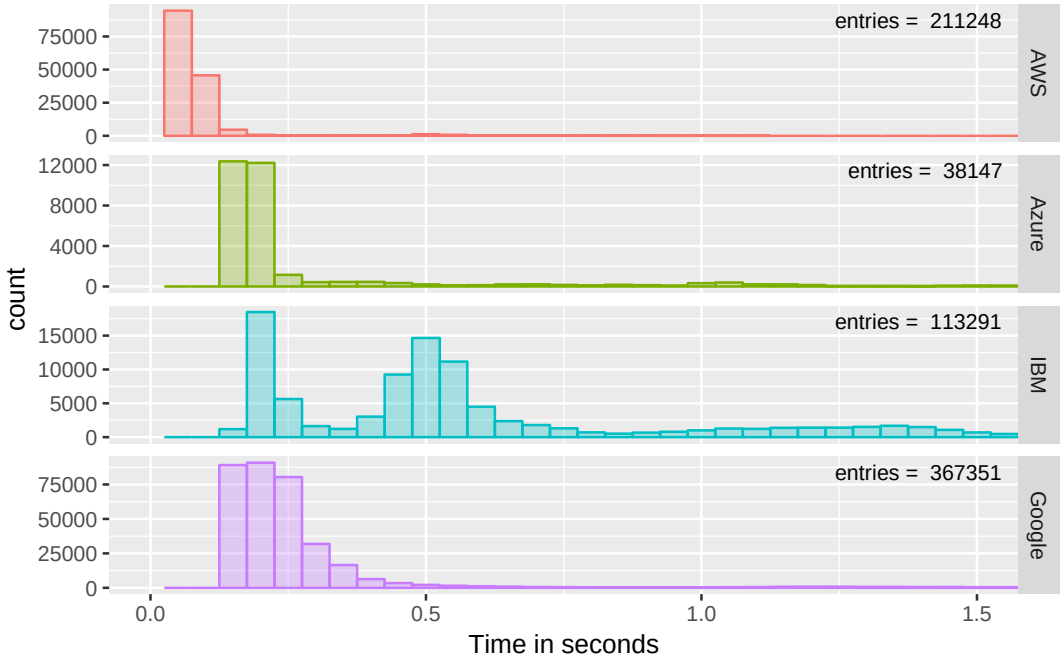


**FIGURE 8** Distribution of $t_o$ overheads for cloud function providers.


**FIGURE 9** Distribution of instance lifetime $t_l$ for cloud function providers.

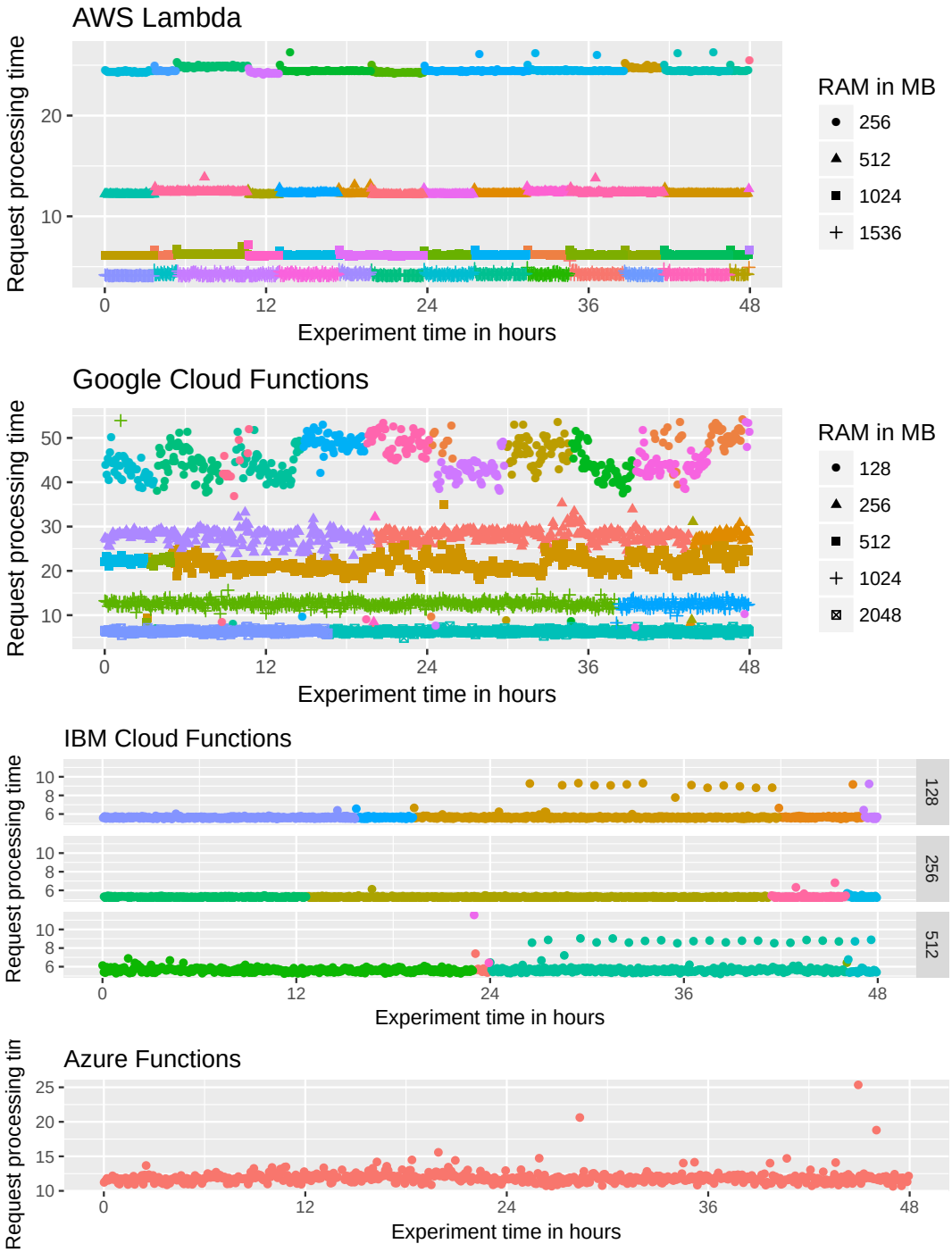


**FIGURE 10** Trace of 48h $t_r$ for cloud function providers. Different colors are used to distinguish different execution environment instances.

Fig. 10 shows the 48 hour trace of $t_l$. One may observe, how environment instances are recycled. We also observe, that there are noticeable changes in mean request processing time for different instances. Also, we can see that the noise level differs – some instances are more noisy than the others.

The analysis of instance lifetime shows also the high heterogeneity of how cloud functions providers implement their services. From Fig. 9 we can see that e.g. AWS restarts their instances regularly (every 2-8 hours), while other providers often leave the processes within the execution environment for a long time, often for several days. We hypothesize that regular reboots of execution environment in the case of AWS Lambda is a

policy implemented with the goal of avoiding performance instability due to software aging, memory leaks, etc. We can expect that other providers will also introduce such policies as their platforms will become more mature and heavily used.

## 6.6 | Cost comparison

Resource consumption costs are critical factors when using cloud computing from public providers in general, and for FaaS services in particular. The pricing models emphasize elasticity and truly on-demand billing models: all the providers charge the CPU usage of a cloud function for every 100 milliseconds of usage, depending on the amount of RAM allocated.

We gathered the current prices of cloud functions from all four providers. As of October 2017 AWS Lambda[7] charges $0.00001667 for every GB-second, Google Cloud Functions[8] cost $0.0000025 for GB-second plus $0.0000100 for GHz-second, Azure Functions[9] cost $0.000016 per GB-second, while IBM Cloud Functions[10] follows the similar price of $0.000017 per GB-second.

In addition to these per-usage prices, for most providers there is also a flat price per request. Moreover, there is always a free tier, for example 400,000 GB-seconds each month are free. In our calculations we ignore these free tiers, since we assume that for large-scale usage they will become negligible. We also do not include the per-request costs, since we are interested in longer-running functions, for which the cost of GB-seconds consumed dominates.

The prices per 100ms billing unit are shown in Fig. 11 , depending on the RAM size. For Azure, we show only the cost of 1GB or RAM, since the user does not have explicit control over memory allocation per function.

We also compare cost of executing single task from our integer performance benchmark in Fig. 12 . For AWS Lambda, we may observe that it does not depend on function size since CPU allocation is proportional to memory allocation. However, on IBM Cloud Functions and Google Cloud Functions, we may observe that smaller functions are cheaper than those with larger memory allocation. The reason is that on IBM platform performance does not depend on function size, while on Google we often get better performance than expected.

To better visualize the price and performance trade-offs, we show the dependency of execution time on the cost of running a given task. The plot in Fig. 13 clearly shows the various pricing policies of the providers and their relative performance.

The cost analysis brings us also some non-trivial results. Looking at the list prices in Fig. 11 , we clearly observe that while all the providers use the same pricing schema and also almost exactly the same prices per MB of RAM, the real costs which take into the account the actual performance vary significantly in Fig. 12 and Fig. 13 . The first important observation is that for AWS the cost of running our task does not depend on the function size: the faster the function, the more expensive it is, but at the same time the execution time is shorter. These effects cancel-out and the price remains almost identical. This means that for CPU-intensive applications it is much more economical to use larger functions, since the price will be the same, but the results will be achieved much faster than when using slower functions. For IBM Cloud Functions the performance does not depend on the function size, which means the cost grows with the size. For Google, the performance is not exactly linearly proportional to the function size, which results in the uneven and non-monotonic cost distribution: the most cost-efficient is the smallest function, while 2048 MB one is still cheaper than 512 and 1024. Such results can make resource management decisions non-trivial, since both execution time and cost have to be taken into account when deciding about choosing the function size for a task. For Azure we do not have the control nor the information on the actual memory consumed, so our cost estimates are approximate. We can observe that the performance of Azure functions was similar to that of Google with 1024 MB, so we can expect that the cost is of similar amount. Relative costs place IBM service favourably compared to others, while on the other hand AWS has the fastest instance.

The results of these price-performance comparisons can provide useful insights for users who want to deploy their compute-intensive functions. We also plan to use them in the future work on scheduling and optimization of scientific applications on these infrastructures.

## 6.7 | Infrastructure heterogeneity

In order to evaluate heterogeneity of underlying hardware infrastructure we report processor model from our integer-based CPU intensive benchmark. We were able to collect data about hardware infrastructure only for AWS Lambda and IBM Cloud Functions by reading /proc/cpuinfo file. Unfortunately, on Google Cloud Functions generic processor is reported. We were also unable to get this information on Azure Functions.

On IBM Cloud Functions there are two processors reported: 65% of requests are handled by Intel Xeon E5-2683 v4 with base frequency of 2.10GHz, while remaining 35% are handled with previous revision of the same model – Intel Xeon E5-2683 v3 (2.00GHz). AWS Lambda runs on more heterogeneous infrastructure. Four processor models were reported in total. 65% of requests were handled with E5-2680 v2 (2.80GHz) while

[7] https://aws.amazon.com/lambda/pricing/
[8] https://cloud.google.com/functions/pricing
[9] https://azure.microsoft.com/en-us/pricing/details/functions/
[10] https://console.bluemix.net/openwhisk/learn/pricing

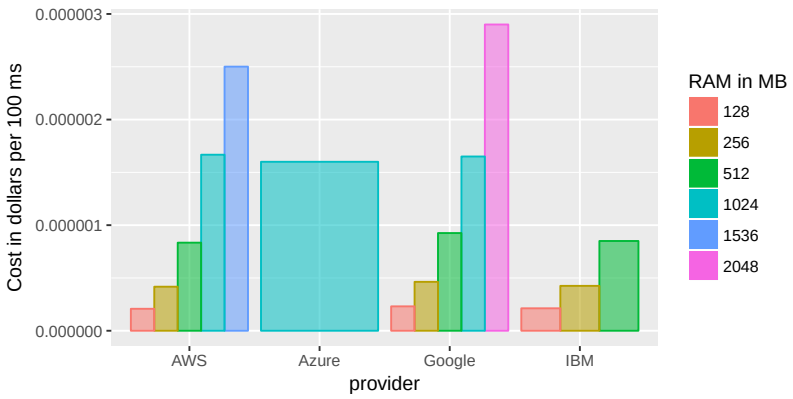


**FIGURE 11** Price for cloud function per 100 millisecond depending on RAM. For Azure we assumed the cost of 1024MB.

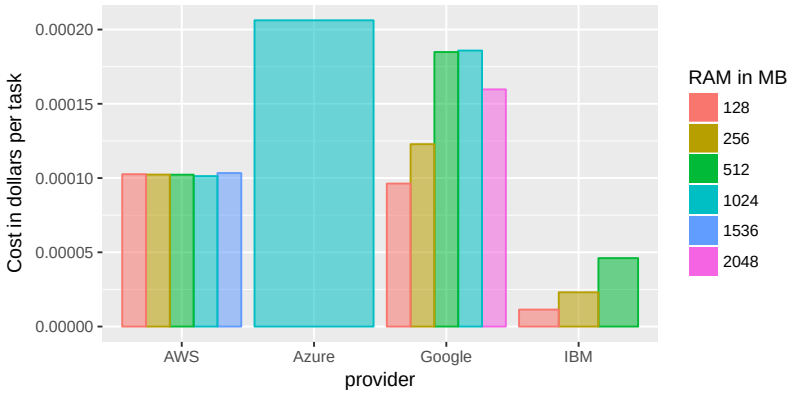


**FIGURE 12** Costs for execution of single task in our integer performance benchmark, for all cloud function providers depending on RAM. For Azure we assumed the cost of 1024MB.

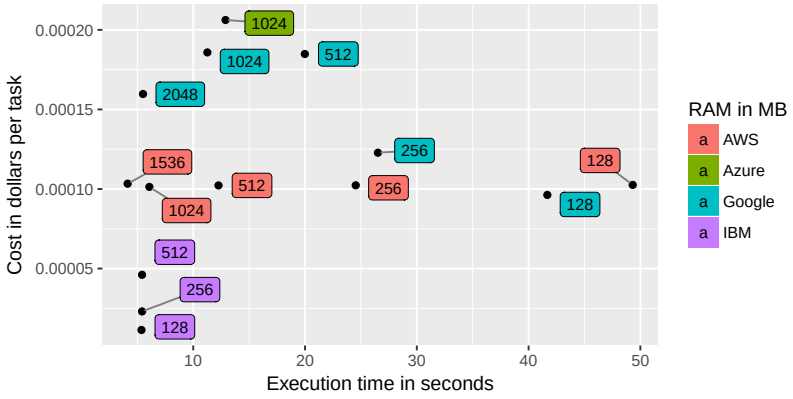


**FIGURE 13** Comparison of cost vs. execution time of single task in our integer performance benchmark, for all cloud function providers depending on RAM. For Azure we assumed the cost of 1024MB.

29% were handled with E5-2666 v3 (2.90GHz). Remaining requests were handled with E5-2676 v3 (2.40GHz) (5%) and E5-2670 v2 (2.50GHz) (1%). Those CPUs support TurboBoost and some cores may be running at slightly higher frequency than base. We did not observe significant correlation between CPU model and function performance though.

## 7 | DISCUSSION OF RESULTS

In this section we recapitulate our hypotheses and summarize how they are verified by our experiments.

Hypothesis 1: **Computational performance of a cloud function is proportional to function size.**

This is true for AWS Lambda and Google Cloud Functions, with the exception of about 5% cases when Google function runs faster than expected. This is not true for IBM and Azure, since the performance does not depend on the function size.

Hypothesis 2: **Network performance (throughput) of a cloud function is proportional to function size.**

This is confirmed for AWS and Google, with the same restriction as hypothesis 1. We have not measured transfers for IBM and Azure, so this still needs to be verified.

Hypothesis 3: **Overheads do not depend on cloud function size and are consistent for each provider.**

This hypothesis was generally confirmed for all the providers.

Hypothesis 4: **Application server instances are reused between calls and are recycled at regular intervals.**

This was nicely demonstrated by our experiments, and we also observed that the instance lifetime differs between providers.

Hypothesis 5: **Functions are executed on heterogeneous hardware.**

While we were able to get results only for two of four providers, we can clearly see that the hardware that runs FaaS infrastructure is heterogeneous.

We have to note that while our hypotheses are generally confirmed, the most interesting observations are those when we see some exceptions or deviations from the general patterns. The specific resource allocation policies as these of Google, or different variances of the results have to be taken into account when making decisions about choosing the provider and the function size. Moreover, the price/performance analysis needs to be carefully performed to avoid unnecessary costs.

There may be also a question about the periodic variations in the performance results or cloud functions. Although we did not perform such detailed statistical tests as in (9), our observations confirm that there is not significant dependency of the time of day or day of week on the cloud providers performance. The existing fluctuations tend to have random characteristics, but it may be subject to further studies once we collect more data.

## 8 | SUMMARY AND FUTURE WORK

In this paper, we presented our approach to performance evaluation of cloud functions. These studies were motivated by the increasing interest of commercial and scientific application of these types of highly-elastic infrastructures. We proposed 5 hypotheses regarding the expected behavior of cloud functions and we designed the benchmarks to verify them. We described our performance evaluation framework, consisting of two suites, one using the Serverless Framework, and the one based on HyperFlow. We gave the technical details on how we address the heterogeneity of the environment, and we described our automated data taking pipeline. We made our experimental primary data available publicly to the community and we set up the data taking as a continuous process.

The presented results of evaluation using Mersenne Twister and Linpack benchmarks show the heterogeneity of cloud function providers, and the relation between the cloud function size and performance. We also revealed the interesting observations on how Amazon and Google differently interpret the resource allocation policies. These observations can be summarized that AWS Lambda functions execution performance is proportional to the memory allocated, but sometimes sightly slower, while for Google Cloud Functions the performance is proportional to the memory allocated, but often much faster. This behavior is also confirmed for data transfer times, which we measured for Google Cloud Functions and AWS Lambda. Our results verified the hypotheses we proposed, and showed their ranges of applicability and exceptions. We believe they can be useful for resource allocation strategies and for planning deployments of serverless applications.

This paper presents the current snapshot of our results of this endeavor, and we continue to add more measurements to our benchmarking suite. In addition to gathering more data, there is also room for other future work. It includes the integration of HyperFlow with our serverless

benchmarking suite, and measurement of influence of parallelism. We consider also possible analysis of trends as we continue to gather more data, as well as implications for resource management.

**Acknowledgements.**

This work was supported by the National Science Centre, Poland, grant 2016/21/B/ST6/01497.

## References

[1] Malawski Maciej, Gajek Adam, Zima Adam, Balis Bartosz, Figiela Kamil. Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions. *Future Generation Computer Systems*. 2017;(In Print).

[2] Balis Bartosz. HyperFlow: A model of computation, programming approach and enactment engine for complex distributed workflows. *Future Generation Computer Systems*. 2016;55:147 - 162.

[3] Malawski Maciej, Figiela Kamil, Gajek Adam, Zima Adam. Benchmarking Heterogeneous Cloud Functions. In: Heras Dora B., Bougé Luc, eds. *Euro-Par 2017: Parallel Processing Workshops, Lecture Notes in Computer Science, vol 10659*, Springer 2018 (pp. 415–426).

[4] Walker Edward. Benchmarking Amazon EC2 for high-performance scientific computing. *LOGIN*. 2008;33(5):18–23.

[5] Berriman G Bruce, Deelman Ewa, Juve Gideon, Rynge Mats, Vöckler Jens-S. The application of cloud computing to scientific workflows: a study of cost and performance.. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*. 2013;371(1983):20120066.

[6] Iosup Alexandru, Ostermann Simon, Yigitbasi Nezih, Prodan Radu, Fahringer Thomas, Epema Dick. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Transactions on Parallel and Distributed Systems*. 2011;22(6):931–945.

[7] Lenk Alexander, Menzel Michael, Lipsky Johannes, Tai Stefan, Offermann Philipp. What Are You Paying For? Performance Benchmarking for Infrastructure-as-a-Service Offerings. In: Proc. , ed. *2011 IEEE 4th International Conference on Cloud Computing*, :484–491IEEE; 2011.

[8] Bocchi Enrico, Mellia Marco, Sarni Sofiane. Cloud storage service benchmarking: Methodologies and experimentations. In: Proc. , ed. *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, :395–400IEEE; 2014.

[9] Leitner Philipp, Cito Jürgen. Patterns in the Chaos - A Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Trans. Internet Techn.*. 2016;16(3):15:1–15:23.

[10] Leitner Philipp, Scheuner Joel. Bursting with Possibilities - An Empirical Study of Credit-Based Bursting Cloud Instance Types. In: Proc. , ed. *8th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2015, Limassol, Cyprus, December 7-10, 2015*, :227–236; 2015.

[11] Prodan Radu, Sperk Michael, Ostermann Simon. Evaluating High-Performance Computing on Google App Engine. *IEEE Software*. 2012;29(2):52–58.

[12] Malawski Maciej, Kuzniar Maciej, Wojcik Piotr, Bubak Marian. How to Use Google App Engine for Free Computing. *IEEE Internet Computing*. 2013;17(1):50–59.

[13] Villamizar M., Garces O., Ochoa L., et al. Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. In: Proceedings , ed. *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016 (pp. 179-182).

[14] Wagner Brandon, Sood Arun. Economics of Resilient Cloud Services. In: Proceedings , ed. *1st IEEE International Workshop on Cyber Resilience Economics*, ; 2016.

[15] McGrath M. Garrett, Short Jared, Ennis Stephen, Judson Brenden, Brenner Paul R.. Cloud Event Programming Paradigms: Applications and Analysis. In: Proceedings , ed. *9th IEEE International Conference on Cloud Computing, CLOUD 2016, San Francisco, CA, USA, June 27 - July 2, 2016*, IEEE Computer Society 2016 (pp. 400–406).

[16] Jacob Joseph C, Katz Daniel S, Berriman G Bruce, et al. Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. *International Journal of Computational Science and Engineering*. 2009;4(2):73–87.

[17] Jiang Qingye, Lee Young Choon, Zomaya Albert Y.. Serverless execution of scientific workflows. In: M. Maximilien , Vallecillo A., Wang J., M. Oriol , eds. *Service-Oriented Computing. ICSOC 2017*, :706–721Springer, Cham; 2017.

[18] Spillner Josef. Snafu: Function-as-a-Service (FaaS) Runtime Design and Implementation. *CoRR*. 2017;abs/1703.07562.

[19] Pérez Alfonso, Moltó Germán, Caballer Miguel, Calatrava Amanda. Serverless computing for container-based architectures. *Future Generation Computer Systems*. 2018;83:50–59.

[20] Varghese Blesson, Buyya Rajkumar. Next generation cloud computing: New trends and research directions. *Future Generation Computer Systems*. 2018;79:849–861.

[21] Castro Paul, Ishakian Vatche, Muthusamy Vinod, Slominski Aleksander. Serverless Programming (Function as a Service). In: Proc. , ed. *International Conference on Distributed Computing Systems,* ; 2017.

[22] Bryan Liston . *Analyzing Genomics Data at Scale using R, AWS Lambda, and Amazon API Gateway | AWS Compute Blog.* http://tinyurl.com/h7vyboo; 2016.