A Framework and a Performance Assessment for Serverless MapReduce on AWS Lambda

V. Giménez-Alventosa[a,*], Germán Moltó[a], Miguel Caballer[a]

[a] *Instituto de Instrumentación para Imagen Molecular (I3M)*
Centro mixto CSIC - Universitat Politècnica de València
Camino de Vera s/n, 46022, Valencia

Abstract

MapReduce is one of the most widely used programming models for analysing large-scale datasets, i.e. Big Data. In recent years, serverless computing and, in particular, Functions as a Service (FaaS) has surged as an execution model in which no explicit management of servers (e.g. virtual machines) is performed by the user. Instead, the Cloud provider dynamically allocates resources to the function invocations and fine-grained billing is introduced depending on the execution time and allocated memory, as exemplified by AWS Lambda. In this article, a high-performant serverless architecture has been created to execute MapReduce jobs on AWS Lambda using Amazon S3 as the storage backend. In addition, a thorough assessment has been carried out to study the suitability of AWS Lambda as a platform for the execution of High Throughput Computing jobs. The results indicate that AWS Lambda provides a convenient computing platform for general-purpose applications that fit within the constraints of the service (15 minutes of maximum execution time, 3008 MB of RAM and 512 MB of disk space) but it exhibits an inhomogeneous performance behaviour that may jeopardise adoption for tightly coupled computing jobs.

Keywords: MapReduce, Serverless, Cloud Computing, Elasticity

1. Introduction

MapReduce [24] is one of the most used programming models for analysing large-scale datasets, i.e. Big Data. Apache Hadoop [3], an open-source platform for reliable, scalable and distributed computing provided the execution support to the MapReduce programming model. Indeed, the Hadoop ecosystem has flourished in the last years resulting in a myriad of tools and services for distributed programming, NoSQL databases, SQL-based databases and machine learning, among many other categories (see the Hadoop Ecosystem table [12] for further details). Apache Hadoop requires a distributed computing infrastructure to support the execution of MapReduce applications.

*Corresponding author: Tel. +34963877356
Email address: vicent.gimenez@i3m.upv.es (V. Giménez-Alventosa)

Infrastructure as a Service (IaaS) Clouds provide distributed computing infrastructures, offered by public Cloud providers such as Amazon Web Services (AWS) [4], Microsoft Azure [6] and Google Cloud Platform (GCP) [11]. Indeed, these providers include in their portfolios services to be able to automatically deploy Hadoop (and Spark) clusters on pay-as-you-go basis. This is the case of Amazon Elastic MapReduce (EMR) [10], for AWS, HDInsight [13] for Microsoft Azure, or Cloud Dataproc [7], for GCP. These services provide automatic deployment capabilities and, to some extent, the ability to scale the clusters in order to increase or decrease the number of worker nodes of the cluster. There are also software projects to dynamically deploy Hadoop (and Spark) clusters on on-premises Clouds. This is the case of Sahara [17], to provision data-intensive application clusters on top of the OpenStack [16] Cloud Management Platform (CMP).

In recent years serverless computing has surged as an execution model in which no explicit management of servers (e.g. virtual machines) is performed by the user. Instead, the Cloud provider dynamically allocates resources to the execution unit, which are function invocations. Users write functions in a programming language supported by the service with no assumption about the underlying computing infrastructure on which the function invocation will be run on. By creating stateless functions, a highly-parallel execution model can be used, where multiple concurrent invocations of the same function can be triggered in response to events occurred in the infrastructure. The cost model for serverless computing really becomes pay-per-use, since no costs are involved for the user unless the function is invoked, as opposed to traditional cost models for IaaS clouds, where Virtual Machines are typically billed per-second regardless of their actual usage. Because of the advantages listed above, serverless computing is now used as an effective simple programming model for a variety of applications [22] [25] [31].

One of the pioneer services of serverless computing is AWS Lambda [5], which allows to execute stateless functions coded in one of the programming languages supported (Node.js, Java, C#, Python and Go), in response to events, on a massive scale (up to 3000 parallel invocations). As opposed to traditional IaaS Clouds, in AWS Lambda there is no explicit management of servers by the users and elasticity is automatically provided by the platform, since the functions are designed with no assumptions on the underlying infrastructure, i.e., they are designed stateless. The large degree of parallelism supported by AWS Lambda is one of the key features of the service.

Indeed, the report from the workshop and panel on the Status of Serverless Computing of the First International Workshop on Serverless Computing (WoSC) 2017 [21] indicated that a hot research topic will be its use for parallel programming and there is a challenge to extend the MapReduce use of FaaS.

To this aim, this paper studies the suitability of AWS Lambda to support the MapReduce programming model and provides two key contributions. On the one hand, an open-source framework is introduced (MARLA - MApReduce on LAmbda) in order to run MapReduce jobs in Python on AWS Lambda. On the other hand, a thorough performance assessment of this service is performed through the execution of several case studies. This highlights that AWS Lambda shows sporadic disparate performance which affects the timeliness of tightly coupled jobs, as is the case of MapReduce applications. We propose identification techniques of these issues and, to the extent that is possible, mitigation approaches to minimise the impact for the execution of these kind of jobs.

After the introduction, the remainder of the paper is structured as follows. First, section 2 describes the related work in the area and compares the existing approaches

with the proposed framework in this paper. Next, section 3 introduces the MARLA framework, describes its architecture and points out the optimisation strategies required to efficiently take advantage of the computing capabilities of AWS Lambda. Then, section 4 provides a detailed assessment of the behaviour of AWS Lambda when executing several workloads. To the authors' knowledge this is the most comprehensive performance study carried out in AWS Lambda that identifies both the benefits and the limitations of this serverless platform as a general-purpose scalable computing platform. Finally, section 5 summarises the main achievements of this paper and points to future work.

2. Related Work

There can be found in the literature few works using serverless computing to support the execution of MapReduce jobs.

AWS has its own Serverless Reference Architecture for MapReduce (SRAM) described in [14]. In this architecture the user previously uploads to an S3 bucket the input data, that has to be previously partitioned in chunks, and write their code as Python or Node.js functions. Then, there is a Python application that the user executes locally in charge of orchestrating the execution by creating the necessary Lambda functions to act as mappers and reducers and starting the execution of the MapReduce job.

Another approach is described in the work by Jonas et al. [18] in which they describe PyWren[1], a platform that enables running Python code at massive scale via AWS Lambda. It enables to launch simple Python parallel programs but also MapReduce jobs. Since it is a general solution, the code required to create a MapReduce application demands a higher complexity compared to MARLA or the SRAM. Indeed, as it happens with the SRAM, it also requires to have the input data pre-partitioned.

Ooso [15] is a serverless MapReduce Java library based on AWS Lambda. The user defines the functions using Java and uploads the pre-partitioned data to an S3 bucket, as is the case of the aforementioned solutions. It relies on Terraform[2] to create all the related AWS resources, instead of Boto, which is commonly used by the previous solutions.

Corral [9] is a framework designed to be deployed to serverless platforms, like AWS Lambda. This framework is entirely wrote in Go. The runtime model consists of stateless, transient executors controlled by a central driver.

As opposed to previous works, MARLA improves previous solutions since it covers all the steps of the MapReduce job execution, including automated partition of the input data using the optimum size for the memory assigned to the Lambda functions. Using MARLA the user only has to define the *map* and *reduce* functions in Python and then use the framework to automatically define the functions. This results in the creation of the Lambda functions and the binding to a certain S3 bucket so that a latent MapReduce service is deployed at zero cost. Then, each time a file is uploaded to that S3 bucket all the activities related to the execution of the MapReduce job are automatically carried out and the results are also written to the output S3 bucket. This provides an effective and convenient approach that provides a highly-scalable MapReduce execution service that does not involve the pre-deployment of computational instances and that can be automatically triggered on-demand in response to a file upload event.

[1] `http://pywren.io`
[2] `https://www.terraform.io/`

MARLA does not require a local application to orchestrate the start of the MapReduce jobs, thus providing a highly scalable multi-tenant MapReduce framework that is triggered on demand. When multiple files (from one or more users) are uploaded to the input bucket on S3, all executions will be processed concurrently, thus avoiding the bottleneck produced by using an external orchestrator.

There are also works that have studied the performance of serverless computing platforms for different Cloud providers (see [28], [27] and [30] for details). Previous studies have shown that AWS Lambda runs on an heterogeneous system and the performance of the Lambda invocations will be proportional to the allocated memory. However, additional tests will be performed in this work in order to determine the performance behaviour of AWS Lambda when running coupled job executions such as those that arise when executing a MapReduce application.

3. Architecture of MARLA

3.1. Architecture

MARLA has been designed as a lightweight framework to allow the execution of Python-based MapReduce jobs on AWS Lamba. To this aim, this section describes the proposed architecture, its components and the workflow to efficiently perform data-oriented processing on such serverless platform.

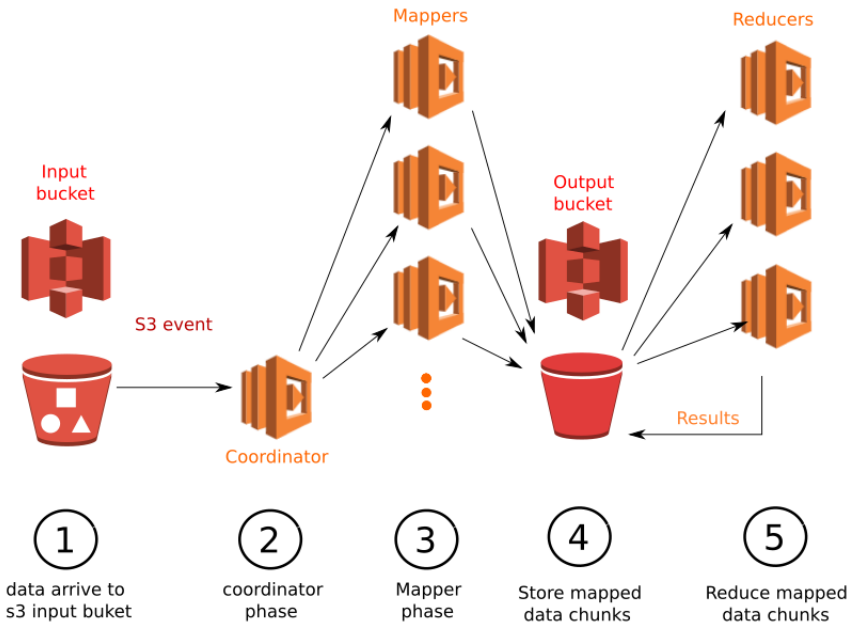

Figure 1: Architecture of MARLA to support MapReduce on AWS Lambda.

Figure 1 shows the architecture diagram. The architecture is entirely composed by Cloud services on AWS without any external components. It consists of three groups

of Lambda functions: The *coordinator*, the *mappers* and the *reducers*. In addition, it requires two S3 buckets, one for input data and another one to collect the post-processed data generated as output.

The execution workflow of the MapReduce job starts when a dataset is uploaded in the "input" S3 bucket. This causes an S3 event to activate an invocation to the coordinator Lambda function, which calculates the optimal size of the data partitions dividing the total dataset size into as many chunks as the user indicates (N) provided that the mapper functions can store in memory that amount of data in their allocated RAM. If the mappers do not have enough RAM, the coordinator Lambda function will recalculate the number of chunks to fit in each mapper.

To calculate the size of the data chunk processed by each mapper, the coordinator function does the following process. First, it calculates the possible size of each data chunk dividing the size of the file to process by the number of mappers specified by the user.

$$chunkSize = \frac{totalDataSize}{N_{mappers}}. \tag{1}$$

Next, the coordinator function checks if the $chunkSize$ does not satisfy the conditions listed below. These conditions are checked in the following sequence:

- $chunkSize$ is smaller than the minimum block size specified by the user ($MINBLOCKSIZE$). In this case, $chunkSize$ value will be set to $MINBLOCKSIZE$ and the number of mappers are recalculated as follows,

$$N_{mappers} = int\left(\frac{totalDataSize}{MINBLOCKSIZE}\right) + 1 \tag{2}$$

- $chunkSize$ is bigger than the calculated safe memory size ($safeMemorySize$). In the coordinator function, $safeMemorySize$ is calculated as a percentage of the memory assigned to the mapper functions. This ensures that the data to process fits in the mapper functions. In this case, the $chunkSize$ value will be set to $safeMemorySize$ and the number of mappers recalculated as follows:

$$N_{mappers} = int\left(\frac{totalDataSize}{safeMemorySize}\right) + 1 \tag{3}$$

- $chunkSize$ is bigger than maximum block size specified by the user ($MAXBLOCKSIZE$). In this case, the $chunkSize$ value will be set to $MAXBLOCKSIZE$ and the number of mappers is recalculated as follows:

$$N_{mappers} = int\left(\frac{totalDataSize}{MAXBLOCKSIZE}\right) + 1 \tag{4}$$

Notice that in all the previous recalculations of $N_{mappers}$, the coordinator function always adds one extra mapper, which is responsible to process the residual data chunk with size:

$$residualData = totalDataSize - (N_{mappers} - 1) \cdot chunkSize \quad (5)$$

Notice that the size of the residual data, in the worst case scenario, will be $chunkSize-1$ *bytes*. This approach prevents any mapper to process its corresponding chunk plus the mentioned residual data, what could cause an additional overload in that mapper. However, depending on the dataset, the residual data can be small enough to create an under-loaded mapper that finishes its execution faster than the other mappers.

Finally, the coordinator invokes the first mapper Lambda function with an environment variable that stores the dimension of each data chunk size. The first mapper Lambda function starts a logarithmic reduction approach so that it invokes the second mapper and, then, both Lambda function invocations will invoke another two. This procedure is recursively repeated until all mappers have been invoked. Notice that this approach shows a $log_2(N_m)$ cost, where N_m is the number of mappers thus minimising, to the extent that is possible, the delays related to the invocation of the mapper function.

Next, each mapper downloads its chunk of data to perform the *map* phase in parallel to the other invocations. This *map* process produces as results a list of *key/value* pairs that will be reduced in the next phase. To process the data, the mapper Lambda function employs the user-defined mapping function.

Then, the mapper Lambda function will sort the data and divide the mapped results in chunks (mapped chunks). Each mapped chunk will be processed by an independent *reduce* Lambda function. We assume that the *reduce* process only needs all *pairs* with the same *key* to complete the data processing i.e. *pairs* with different *key* are processed independently in the reduce step. This is not a hard restriction since the user can assign any *key* to mapped *pairs* during the *map* process. Therefore, to ensure that each *reduce* Lambda function has all the required data for post-processing, the division to create mapped chunks is performed given the first character of each *pair key* obtained from the previous mapping process. Therefore, the generated mapped chunks for each reduce will be alphabetically independent, i.e. all pair *keys* with the same first character will be stored in the same mapped chunk, ensuring that the mentioned condition for the *reduce* step will be fulfilled.

These chunks are stored in the output bucket with a name that identifies their corresponding reduce number and the mapper that produced it. In addition, a prefix for each stored mapped chunk will be added using an *md5* hash to achieve optimal S3 performance, as described in [20]. This last step ensures a good distribution of all data chunks among S3 partitions, thus enhancing the data access performance in S3.

It is important to point out that before the execution of the mapper Lambda function corresponding to the penultimate partition ends, it will invoke the first reduce Lambda function, that will perform a tester function. This single reduce will check the uploaded chunks until all mapped partitions corresponding to the last reduce have been uploaded. Then, this function will invoke all the reducers and finish its execution. Notice that, unlike mappers, the number of used reducers on a single execution will be small (in the order of tens). Using more reducers will cause that several reducers try to process the same ASCII interval, so these "duplicate" reducers will not perform any job. In addition, as it will be explained in section 4, incrementing the number of reducers will be counterproductive. Therefore, there is no need to use a logarithmic approach to perform the reduction phase.

In the last step, each Lambda reduce function will execute the user's predefined reduce function on its corresponding mapped data chunks. To do this, it will download as many mapped chunks as possible per the memory size, which is chosen by the user. The reducer will download at least one entire chunk, since a single mapped data chunk will not be partitioned. Then, the reduce user function is executed on the downloaded mapped chunks, plus the previously processed ones. This download-process loop will be repeated until all corresponding mapped chunks have been processed. Therefore, all the mapped chunks are reduced. Finally, the results are stored in the output bucket. Notice that each reduce will produce a results file with its identifier (from 0 to $N_{reduces} - 1$). MARLA has been deliberately designed to not aggregate the results of the multiple reducers, since the memory of a Lambda function is limited to $3008MB$ and this might not be enough to host the final result. So, in each execution, the user will obtain as many result files as reducers used, even if all the result files fit in a single Lambda function. Notice that the aggregation of these results can be performed as an additional post-processing on a high performant EC2 instance. In particular, this aggregation of results is not automatically performed by MARLA, since this is the same behaviour seen in the Apache Hadoop framework, where multiple reducers will generate separate output files and, therefore, the user is responsible for joining these files.

This approach allows to process huge results data files unless the *pair keys*, obtained from the mapping process, were poorly distributed alphabetically. An easy way to distribute the load across multiple reducers if the *keys* are not well distributed in the input data, is to hash the *keys* using numbers and extract the residual part of the integer division with the total number of reducers, as shown in Eq. 6.

$$reduce = hash(key)\%N_{red}, \tag{6}$$

This number assigns an ASCII interval and, consequently, the reducer that must process this *pair key/value*. Next, taking into account that the printable ASCII characters interval (with no ASCII extensions) is $[32, 126]$, the total number of printable characters is $126 - 32 + 1 = 95$. Therefore, the number of characters in each reducer interval can be calculated by,

$$\Delta interval = int\left(\frac{95}{N_{red}}\right) \tag{7}$$

Finally, a character within the corresponding interval must be added as prefix of *pair key* as shown in Eq. 8

$$key = chr\left(\Delta interval \cdot reduce + 33\right) + str(key). \tag{8}$$

Notice that adding 33 and not 32 avoids appending a *space* that will be removed by strip functions. Also, this requires to avoid an overlap with the special characters employed to separate columns in the original dataset. For example in CSV (comma-separated values) files, the comma should not be used as part of the data to avoid misleading interpretations by the framework.

3.2. Failure handling

This section briefly explains failure handling on MARLA. The exceptions caused by bad input from the user will terminate the Lambda function invocation, thus halting the

data analysis. A failure on one or more functions (coordinator, mappers or reducers) will never produce an unlimited number of retries. Indeed, all the Lambda functions will eventually finish, since the "tester" reduce can invoke itself a limited number of times to avoid infinite recursive invocations in case one mapper fails. The same behaviour will be exhibited when a failure is detected uploading or downloading from S3, since the current or next processing step will not find the required data. Depending on where the failure is detected, the behaviour of MARLA will be different:

- **Failure on the Coordinator**: No mapper will be invoked. The execution will end up at this point.
- **Failure on the Mapper function**: Failed mappers will not upload any mapped data to S3, so the tester reduce function will never find all the required partitions and the reducers will not be invoked. Eventually, the reducer tester will finish its invocation. However, successfully executed mappers will upload their mapped data partitions to the output bucket.
- **Failure on the Reduce function**: Even if one or more reducers fail, this only affects to the data that failed reducers must process, since reducers are independent from each other. Therefore, at the end of the execution, the data of successfully executed reducers will be uploaded to S3.

Mapper and reducer failures, in most cases, can leave partially processed data in the S3 output bucket, to be handled by the user.

3.3. Assumptions and Limitations

The current assumptions and limitations of MARLA are described below:

- MARLA works, at the moment, with plain text column-based data files where the column change is specified by a "," character. Future works includes extending support to binary data and other data formats.
- The input data to process does not need to be pre-partitioned. MARLA will perform the data partition step automatically considering the configuration specified by the user and the memory allocated to functions.
- The user "mapping" and "reduce" functions must be implemented in Python, which is the only supported language at the moment.
- The access credentials and IAM roles required to use AWS services must be provided by the user. MARLA will not create any role or permissions, but use the existing ones.

3.4. Assessment of MapReduce on AWS Lambda

This section provides a thorough assessment of the benefits and limitations of AWS Lambda as a general computing platform and, in particular, for the execution of dependent parallel jobs, as in MapReduce applications. It also aims at comparing the developed framework to other data processing frameworks, in particular the Serverless Reference Architecture for MapReduce (SRAM) developed by AWS. To achieve a fair comparison,

we use the same benchmark as the SRAM [14], a subset of the Amplab benchmark [2]. The benchmark measures response time on a handful of relational queries: scans, aggregations, joins, and user-defined functions, across different data sizes. The authors extract workloads and queries from the previous study by Pavlo et al. [23]. However, since the datasets used are different, the results of both benchmarks are not directly comparable.

To be able to compare with the SRAM, the same subset of Amplab benchmark queries will be tested. The queries are:

- **Scan query**: The corresponding data set contains 90 M rows and approximately $6.36GB$ of data. The queries select the *pageURL* and *pageRank* from rankings where $pageRank > X$, where $X = \{1000, 100, 10\}$. The first two queries have been tested.
  - **Scan 1a**: SELECT *pageURL*, *pageRank* FROM rankings WHERE $pageRank >$ 1000.
  - **Scan 1b**: SELECT *pageURL*, *pageRank* FROM rankings WHERE $pageRank >$ 100.
- **Aggregation query**: Involves the *UserVisits* dataset, which contains $775M$ rows with approximately $127GB$. Only one query from this group has been performed, defined as follows:
  - **Aggregate 2a**: SELECT SUBSTR(*sourceIP*,1,8), SUM(*adRevenue*) FROM *uservisits* GROUP BY SUBSTR(*sourceIP*,1,8).

The SRAM only run the subset of queries explained above because the benchmark is designed to increase the output size by an order of magnitude for the a, b, c iterations. Given that the output size, when the reference architecture executed the benchmark, did not fit in the Lambda memory (maximum of $1536MB$ at that time), they could not proceed to compute the final output.

The results obtained by the reference architecture, and other analysis frameworks, are shown in Table 1. This results, extracted from [14], provide the execution times spent to process the queries with different technologies. The results will be briefly explained below. A detailed explanation of the results, together with the used configurations, is available in [2].

Technology	Scan 1a	Scan 1b	Aggregate 2a
Amazon Redshift (HDD)	2.49	2.61	25.46
Impala - Disk - 1.2.3	12.015	12.015	113.72
Impala - Mem - 1.2.3	2.17	3.01	84.35
Shark - Disk - 0.8.1	6.6	7	151.4
Shark - Mem - 0.8.1	1.7	1.8	83.7
Hive - 0.12 YARN	50.49	59.93	730.62
Tez - 0.2.0	28.22	36.35	377.48
Serverless MapReduce	39	47	200

Table 1: Execution time in seconds of each query for different technologies. *Mem* and *Disk* refers to storage type, memmory or disc respectively. This table has been extracted from the Serverless MapReduce reference architecture repository [14].

The first queries (*Scan 1a* and *Scan 1b*) test the throughput obtained with each framework when reading and writing table data. The best performers are Impala (Mem) and Shark (Mem) which show excellent throughput by avoiding disk usage. For on-disk data, Redshift shows the best throughput, since it uses columnar compression which allows it to bypass a field that is not used in the query.

The last query (*Aggregate 2a*) applies string parsing to each input tuple, then performs a high-cardinality aggregation. Redshift's columnar storage provides greater benefit than in both aforementioned queries since several columns of the data table are unused. While Shark's in-memory tables are also columnar, the bottleneck relies on the speed at which the *SUBSTR* expression is evaluated.

For our tests we use 100 mappers Lambda functions of ($1536MB$) for *Scan* queries. However, in the third query, we use the current most performant Lambda functions ($3008MB$) to reduce the number of parallel invocations. For example, for Lambda functions with $1536MB$ we need, at least, 800 parallel invocations to fit all the data. Also, since the queries are designed to increment the results size in one order of magnitude, the number of reducers used in the *Scan* queries are, in order, 1 and 10, since the results of the first query fit in one Lambda function. For the third query (*Agregation1*) we use different configurations to find an optimal one, as we will see below.

First, we show the execution times measured for the *Scan 1a* and *Scan 1b* queries on Figure 3.4. The figure shows important discrepancies in the execution time. Therefore, each query has been repeated 30 times to provide a time distribution instead of obtaining a single value, which definitely provides a misrepresentation of the actual execution time.

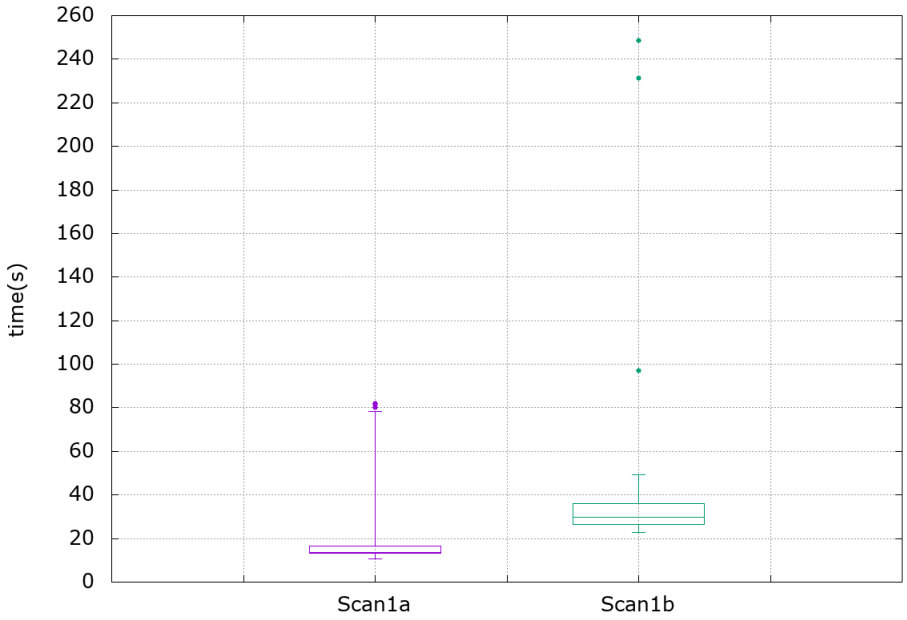

Figure 2: Execution times distribution for *Scan 1a* and *Scan 1b*. Whiskers group the 90% of the data points. For *Scan 1a*, the execution times interval is, in seconds, $[10.699, 81.874]$, with a mean value of $13.653s$. In the same way, the interval of *Scan 1b* is $[22.758, 248.612]$ with a mean value of $29.635s$.

As we can see in Figure 3.4, the measured execution times show significant fluctuations. One of the possible causes could be that some function invocations exhibit a failed S3 request and the entire system waits until this function is restarted in order to finish processing its corresponding data. This kind of errors are explained in the work by Garfinkel [26]. Other possible cause of these fluctuations is the *cold start*, which is the increased invocation latency experienced when a Lambda function is invoked for the first time or after not being used for a certain period of time. This has been extensively studied in the work by Lang et al. [29]. They measured, among other things, cold start times for Node.js 6 Lambda functions of $128MB$ and $1536MB$, shown in Table 2.

Provider-Memory	**Median**	**Min**	**Max**	**STD**
AWS-128	265.21	189.87	7048.42	354.43
AWS-1536	250.07	187.97	5368.31	273.63

Table 2: Cold start latencies (in ms) in AWS using Node.js 6 functions. Data obtained from [29].

However, according to the values in Table 2 the slower executions when executing *Scan 1a* and *Scan 2a* cannot be produced by cold start. In order to find the causes of those fluctuations, additional insights should be obtained from the execution of the MapReduce jobs on AWS Lamba. Figures 3 and 4 show the distribution of the execution time for all mappers in queries *Scan 1a* and *Scan 1b* respectively, across 30 executions. These times have been measured using the execution time information provided by *CloudWatch* logs.

The whiskers were set to group 98% of the data points. Therefore, the two more divergent mappers are represented as points. Notice that the lowest execution time for each run corresponds to the last mapper, who has a residual data chunk, as we saw in section 3. Thus, this low execution time is not an unexpected result. However, we should expect a similar execution time among the other mappers, which process the same amount of data.

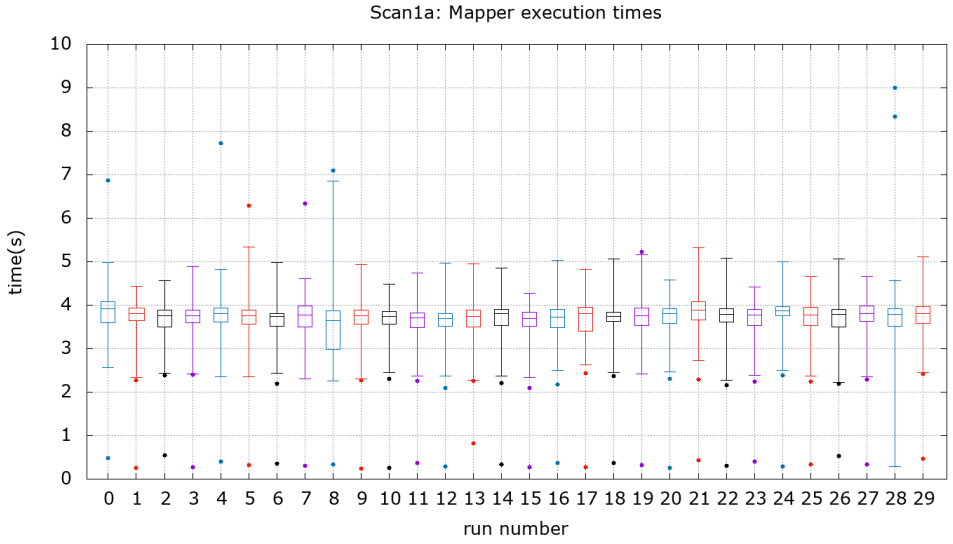

Figure 3: Mapper execution times distribution for the "Scan 1a" query. Whiskers group 98% of total data points. Coloured lines are only included for the convenience of the reader.

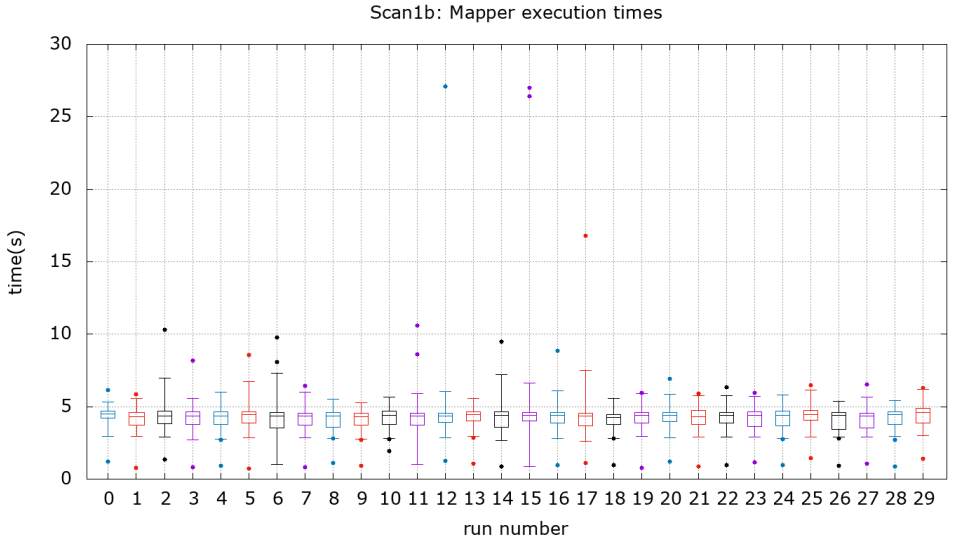

Figure 4: Mapper execution times distribution for the "Scan 1b" query. Whiskers group 98% of total data points. Coloured lines are only included for the convenience of the reader.

The previous figures point out that the execution times of the mappers exhibit a non homogeneous behaviour, showing large fluctuations. On the one hand, *Scan 1a* has only 1 reduce, and its execution times are between 7.8 s and 15.1 s with a mean value of 11.65 s. Therefore, the executions with times of approximately 80 s cannot be exclusively attributed to the cold start. On the other hand, focusing on the *Scan 1b* query, the cold start again cannot explain the high execution times of approximately 240 s shown in Figure 3.4, as it happens with *Scan 1a*. We used 10 reducers in this query, thus improving the execution time of the reducers, since each one processes less data than if there was only one reducer. However, in some executions we observe that few reducers spend more than 100 s to download and process all data, when the mean is 14.675 s.

An analysis of the AWS *CloudWatch* logs revealed that these slow reducers require much longer time than the mean of reducers to download data from S3. This effect seems to be caused by the increment of the number of reducers. The more reducers are used, the more partitions needs each mapper to upload to S3 (one per reducer) and more simultaneous PUT/GET requests will be performed by the Lambda functions. Remember that we adopted a hashed prefix included on uploaded mapped data chunks in order to minimize the scalability limitations in S3 by the underlying data partitions.

Concerning the third query (*Aggregate 2a*), some preliminary tests have been carried out to determine a good configuration to perform the tests. The best configuration tested is 250 mappers and 15 reducers. To determine when a configuration is good enough, we compared the execution time with the one obtained by the SRAM, shown in Table 1. If the first executions on same configuration give execution times worse than those from the SRAM we changed the configuration. For this reason each configuration has a different number of executions and the best one has been repeated 20 times. This result, shown

in Figure 5 together with other configurations, seems to indicate that for these kind of applications that use S3 as temporary storage across stages of the MapReduce, AWS Lambda functions face certain scalability limitations.

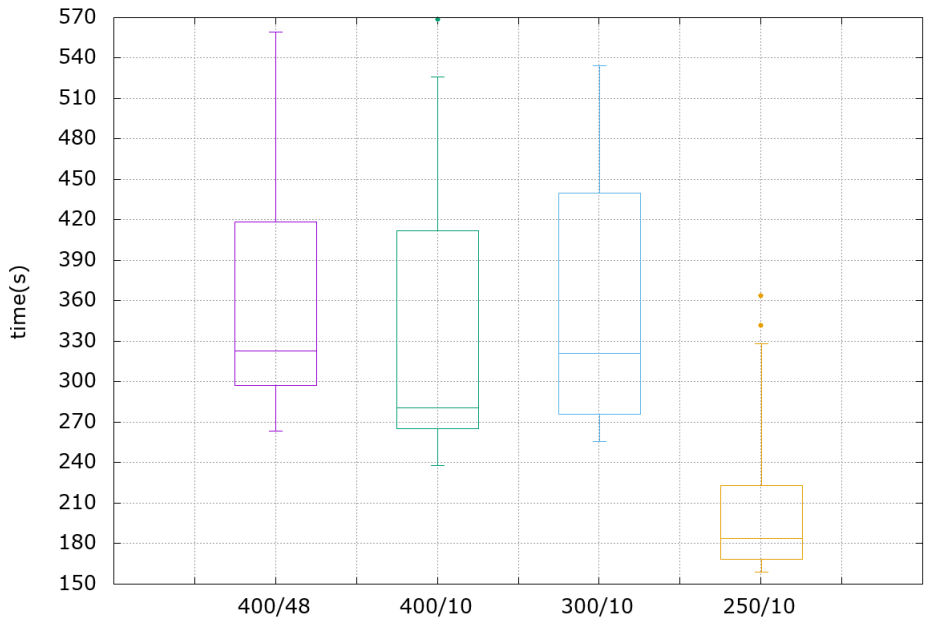

Figure 5: Execution times distribution for "Aggregate 2a". The x axis indicate the number of mappers and reducers used, using the format $N_{mappers}/N_{reducers}$. The number of executions done for each test are, respectively, $6, 10, 5, 20$. Whiskers group the 90% of the data points.

The results shown in this section seem to indicate that AWS Lambda does not provide a homogeneous computing environment even if all Lambda functions have the same memory assigned. However, the previous tests cannot ensure that all functions have the same workload or discriminate the real effect that slow down some of the function invocations. To address these issues, the following section will perform a set of controlled tests in order to investigate the performance discrepancies in AWS Lambda.

Also, the results provided by the SRAM (Table 1) are comparable to our execution time results. However, as we see in the current section, the results in Table 1 should not be interpreted as a precise value. Instead, a range of execution times should be provided, as we have done in this paper, in order to facilitate comparison among researchers. The upcoming section 4 will study the causes of these fluctuations in terms of CPU and network performance.

4. AWS Lambda Performance Assessment

In this section, the suitability of Lambda functions to perform general-purpose computing will be studied. For this purpose, we select two kind of tests: CPU performance and network throughput. Even though the technical implementation details of AWS

Lambda are not disclosed, we anticipate that the performance of a certain Lambda function invocation is related to the performance of the underlying computing node (i.e. Virtual Machine, or instance in AWS terminology) on which it is running. Therefore, in order to identify on which node each Lambda function invocation is running we take advantage of the trick explained in the work by Wang et al. [29], summarised as follows. The */proc/self/cgroup* file has a special entry called *instance root ID* that starts with "sandbox-root-" followed by a 6-byte random string. The authors found that this ID can be used to reliably identity a host virtual machine (VM), even outperforming the heuristic initially proposed by Lloyd et al. [30] based on the VM uptime in `/proc/stat`, which proved to be unreliable.

Another important data to extract is the memory assigned to the VM. To know this value will allow us to calculate the maximum number of Lambda functions that fit in a single VM, depending on the Lambda memory size. To extract that data, we use the command $free - t - m$. The maximum measured VM memory sizes are $3757MB$ and $3945MB$.

All the following tests have been performed on the *us-east-1* region of AWS using Python 3 as the programming language for the Lambda functions. Another factor to consider is that the time has been measured using the Python builtin library *time*, that is, times have been measured inside the Lambda executed code. This approach ensures that the cold start does not influence time measures.

In following subsection, only one Lambda function has been created for each performed test and this one will be invoked several times in multiple test executions, as will be explained then. So, when we talk about invocations, we are referring to invocations of the same test Lambda function.

4.1. CPU performance

For the execution of MapReduce jobs and coupled job executions such as workflows, a desirable property is performance homogeneity or, at least, to be able to anticipate the performance of each node to efficiently distribute the workload. According to AWS documentation, a Lambda function performance is proportional to its user-configured memory [8]. Previous tests in the literature [29] obtained a linear regression between CPU percentage assigned to function and Lambda function memory. However, in these tests, we aim at determining if all Lambda function invocations, with the same assigned memory, have a similar FLOPS performance. Since all of them have the same assigned memory, theoretically, the same CPU performance should shown by each Lambda function. This way, the differences between execution times by executing the very same code must be produced by the underlying heterogeneity in the system.

The tests consist in executing a fixed number of floating point products. The number of operations depends on the Lambda function memory. Therefore, all the invocations with the same assigned memory will do exactly the same amount of computation. Notice that only the time differences among the function invocations is important, regardless of the total execution time.

The *test execution* represents the simultaneous N invocations of one test Lambda function, with the same code and characteristics (memory assigned, timeout, etc.). As we explain before, all the invocations in the same test run the same Lambda function. A test is formed by a predefined number of test executions where each one has identical configuration to each other (same Lambda function, number of invocations, code,

memory, etc.). In addition, test executions will be invoked at regular time intervals that depends on the total workload, to avoid overlapping executions.

For each test execution, each function invocation has a unique numeric identifier (invocation ID or invocation number) starting from 0 to $N-1$, where N is the number of simultaneous invocations in each test execution. To minimize the delay between function invocations, the ID 0 is invoked first, in each test execution. Then, this function invocation is in charge of invoking all the others to prevent, to the extent that is possible, additional delays from our possible network fluctuations, latency and cold start. Because that, the function invocation with ID 0 is not considered in any test execution.

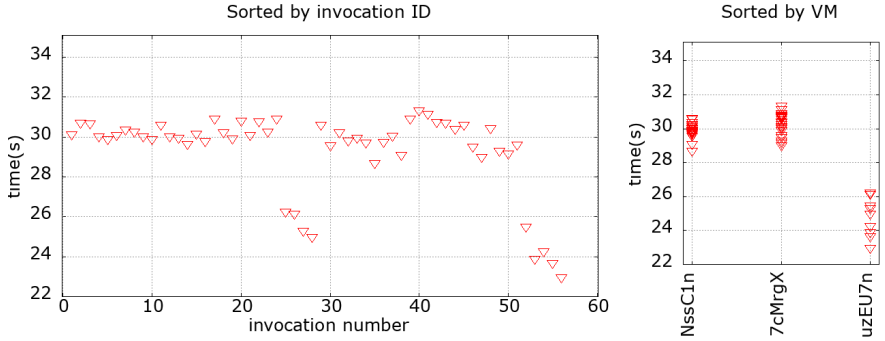

Figure 6: Execution time distribution with 56 invocations of test Lambda function with $128MB$ of memory. Invocations sorted by ID (left) and grouped by virtual machine identifier (right).

In the first test, 56 invocations with $128MB$ of assigned memory have been used. The selected number of Lambda invocations ensures that more than one VM will be used to allocate all functions, taking into account the maximum assigned memory to each VM. The results of one test execution are described in Figure 6, where the execution time of each Lambda invocation has been represented sorting the Lambda invocation by number (left) and grouping invocations that run on the same VM (right). The graph clearly classifies the output results in two different groups that identify two different performance baselines. This proves that AWS Lambda features a heterogeneous computational infrastructure even if the the same memory size is allocated to the Lambda functions. In addition, there is a clear correlation between the execution time of the Lambda invocations and the underlying allocated VM.

We were able to replicate this behaviour across multiple test executions. From the outcome of tests it is possible to claim that the performance delivered by each VM does not change during successive test executions. This fact suggests that this difference of speed can be related with the CPU of the underlying VM on which the function code is being executed (encapsulated in a container). Therefore, we can infer that assigning a certain amount of RAM to a Lambda function indeed sets a certain CPU share in the underlying container run on the EC2 instance (managed by AWS and completely hidden from the user) and, therefore, this share results in greater performance on instances (VMs) with better CPU capabilities.

Additional tests were carried out with different assigned memories and the same behaviour described above was reproduced. For example, Figure 7 shows a test execution of 40 invocations with $1216MB$ of assigned memory. In that test, the assigned VM can

store only two Lambda functions each one. In Figure 7 (down), most VM IDs have two assigned data points corresponding to two invocation measured times. However, the measured times on the same VM are so close that the points are overlapped.

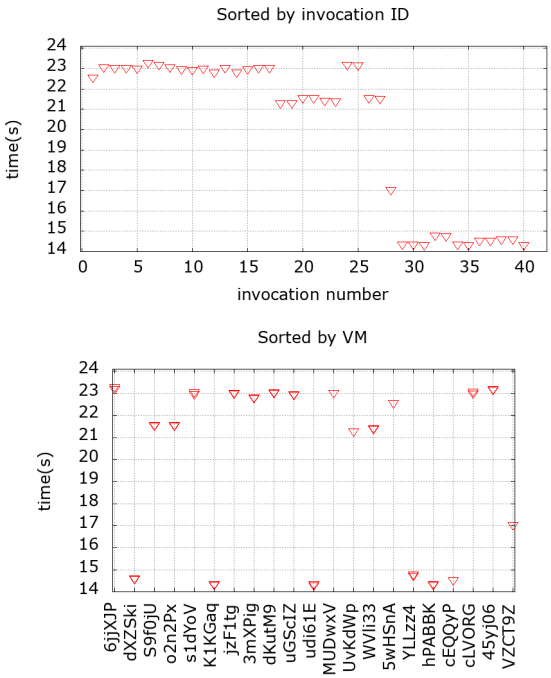

Figure 7: Test with 40 invocations of test Lambda function with $1216MB$ of memory. Invocations sorted by ID (up) and grouped by virtual machine identifier (down).

As mentioned above, this behaviour could be explained if the CPUs used in each VM are not the same, and therefore, deliver different performance. Therefore, in the following test we attempt to find a correlation between CPU model and execution speed. To extract the CPU information of the corresponding underlying VM, the Lambda functions read it from "$/proc/cpuinfo$" file. This test performs 15 test executions with 200 invocations of $1216MB$ each one. Lambda execution times have been classified by VM CPU model. Like previous tests, all Lambda invocations execute exactly the same code. Figure 8 shows a box-whiskers plot where each box color corresponds to one CPU model. In addition, we check that all VMs have only one kind of CPU model and all CPUs of the same model work at same frequency, according to the information extracted from "$/proc/cpuinfo$".

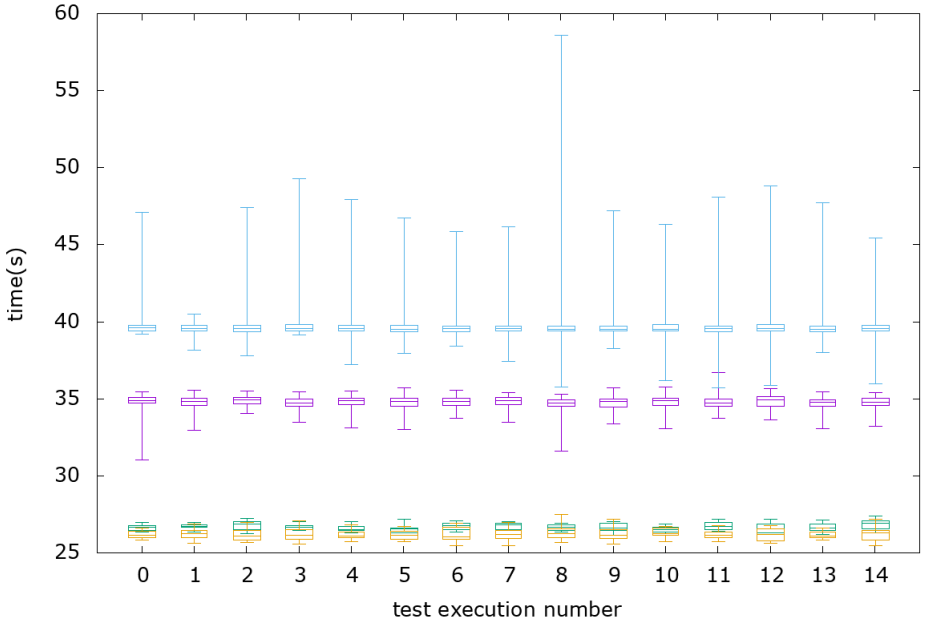

Figure 8: Distribution of execution times for Lambda test invocations grouped by CPU model for each test execution. 200 invocations with $1216MB$ have been used in this test. Each color corresponds to one CPU model, Intel(R)-Xeon(R)-CPU-E5-2666-v3-@-2.90GHz (purple), Intel(R)-Xeon(R)-CPU-E5-2676-v3-@-2.40GHz (green), Intel(R)-Xeon(R)-CPU-E5-2680-v2-@-2.80GHz (blue), Intel(R)-Xeon(R)-CPU-E5-2686-v4-@-2.30GHz (yellow).

Even though Figure 8 shows long whiskers on "slow" CPU models, the boxes are narrow. Therefore, the CPU model is a good indicator of VM performance and, therefore, of the Lambda function invocation performance. However, in "slow" CPU models, we must assume a small probability of big fluctuations on performance. Notice that, far from being a homogeneous system, AWS Lambda functions show performance differences between invocations of same function ones up to a factor of two, in the worst cases.

4.2. Simultaneous network usage performance

Most general-purpose applications executed on AWS Lambda functions require to store partial results on a data storage facility. Since Lambda functions are allocated an ephemeral disk storage space of 512 MB, the best option is to rely on Amazon S3 to store the output data of the Lambda functions.

For this reason, we tested the network performance and fluctuations when multiple Lambda instances simultaneously access S3. Notice that, in the following tests we force Lambda invocations to upload data simultaneously to the same bucket in S3. This can be considered a worst-case scenario since, in general, not all functions will upload data at the same time. Therefore, the effects identified in this section are expected to be smaller under real-world scenarios.

The test includes also the required time to write data to Amazon S3. However, since all Lambda invocations will upload the same amount of data and use a hash as

S3 prefix to ensure good distribution within S3 partitions, the time spent to write data should be similar for all the requests. Notice that adding an approximately constant value to all the measured times cannot provoke significant fluctuations on the measures. On the other hand, AWS ensures that *"your application can achieve at least 3,500 PUT/POST/DELETE and 5,500 GET requests per second per prefix in a bucket"* [1]. Our test uses a single prefix per PUT request and much less requests. Therefore, we assume that S3 is not a bottleneck and does not introduce fluctuations on upload times.

The tests consist on multiple simultaneous executions of the same Lambda function performing two steps. First, it measures the time required to upload dummy data chunks of $500KB$ each one. The number of data chunks (N_{chunks}) will change in each test to vary the workload. Then, big chunks with size of $500 \cdot N_{chunks}KB$ will be uploaded. The *big chunk* upload step forces all Lambda invocations to still use the network when they have already uploaded all chunks. Thus, this step tries to maintain the network usage while Lambda invocations perform the *multi-chunk* upload. This approach is followed to determine whether network saturation occurs. To this aim, it is desirable that all Lambda invocations simultaneously upload data. Notice that only the time dedicated to upload the data will be measured, i.e. the cold start delay is not included in these results, as it was done in the tests performed in section 4.1. We adopt the same approach done in the previous section, where all instances will be invoked by the execution with identifier 0 to mitigate our network fluctuations, latency and cold start during invocations.

To ensure a good partitioning of uploaded data in S3, we followed the recommendations explained in [20]. Therefore, each chunk has a unique prefix generated using an MD5 hash to ensure that the uploaded data is well distributed in the underlying S3 partitions. We used normal upload because *multi-part* upload method requires a minimum size of $5MB$ for each partition, except the last one, as AWS explains in [19]. So, even if all chunks form a unique results file, we cannot use *multi-part* upload for small parts.

All used Lambda functions were assigned $3008MB$ of RAM, the largest value available to date, to ensure that only one invocation of test Lambda function is executed in each VM. As in previous section, for each test, all Lambda invocations run the same Lambda function with same characteristics and code. In each test, we change the number of concurrent Lambda invocations and the number of chunks to upload to modify the workload.

Figure 9 shows the results of a test with 10 test executions using 400 function invocations and 40 chunks. The whiskers of the box plot group 98% of the data while the rest of data are represented as points. The figure shows that most of the invocations conclude the upload step fast, but a very small fraction take a lot of time to finish the task. In addition, as we can see on different test executions, the time required by the slower invocations seems to be unpredictable. On successive test executions, we checked that the slow invocations were not executed in the same VM. Some VMs host "fast" invocations on some executions and "slow" on others. Also, seeing the upload time spent in the "big chunk" step, one can observe that some slow invocations upload data at regular speed in this step. Probably this was because most Lambda invocations have finished all their uploads. Therefore, unlike the case described in subsection 4.1, this increment in upload time seems unrelated to the performance of the VM but still has a major impact in the global execution time.

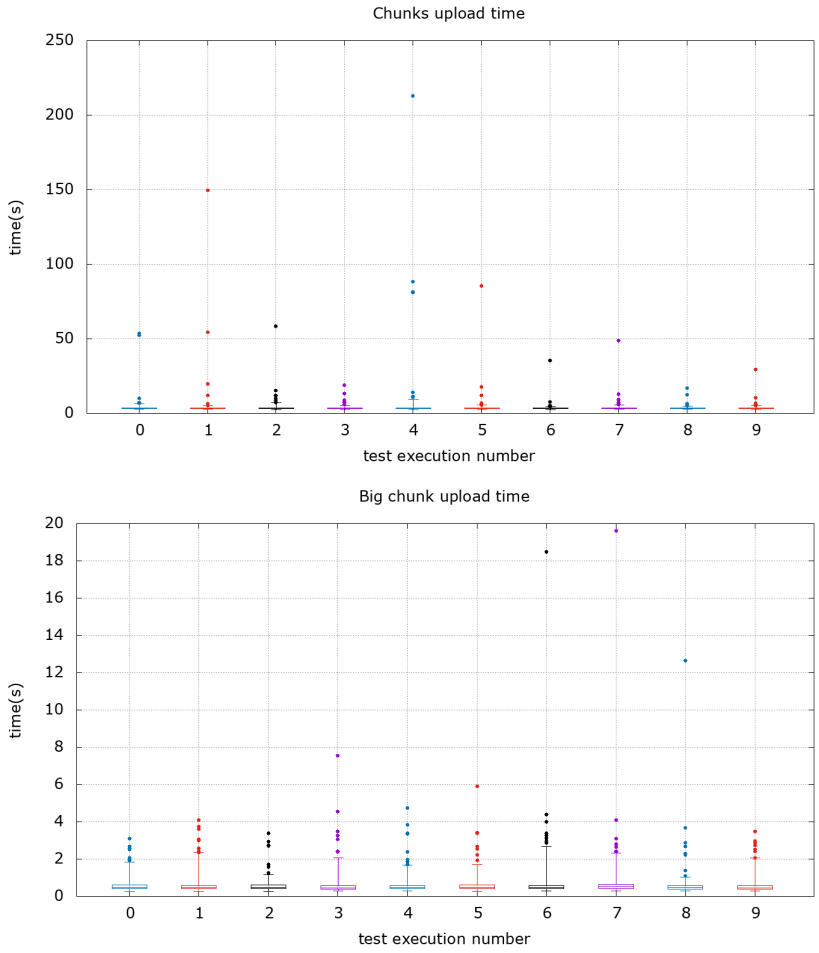

Figure 9: Network test using 400 Lambda invocations with $3008MB$ and 40 chunks. 10 test executions were performed. The plots show distributions of time required by Lambda invocations to conclude the small chunks upload step (up) and the "big chunk" upload step (down). Whiskers group the 98% of data. Coloured lines are only included for convenience.

The previous figure shows that the network behaviour has unpredictable fluctuations in a small subset of invocations. This fact shows that the slow network effect is not caused by a slow VM, but is caused by some kind of network saturation or resources assignation to each Lambda invocation. A similar behaviour can be observed testing the download time.

Several tests have been carried out using different number of concurrent invocations and chunks. Next, we show two tests: one with 100 invocations and 20 chunks and other with 20 invocations and 20 chunks. Notice that test functions have similarity with the mapper functions in the MARLA architecture. The number of test function invocations would be the number of mappers and the number of chunks would be the number of reducers. So, we select the number of function invocations and chunks in each test to be

reasonable values for our architecture while testing different workloads.

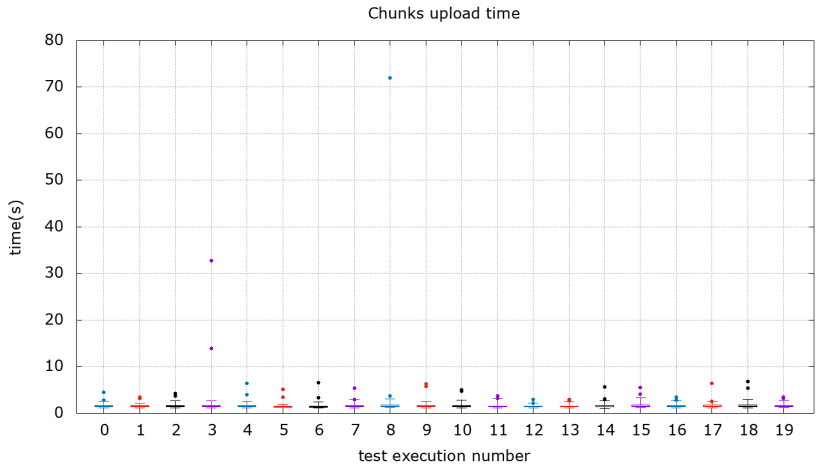

Figure 10: Network test using 100 Lambda invocations with $3008MB$ and 20 chunks. 20 test executions have been performed for this test. Plots show distributions of time required by invocations to conclude the small chunks upload step. Whiskers group the 98% of data.

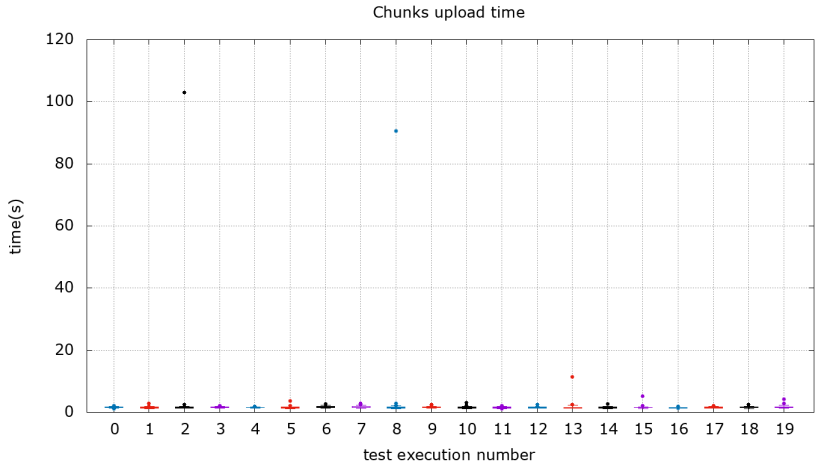

Figure 11: Network test using 20 Lambda invocations with $3008MB$ and 20 chunks. 20 test executions have been performed for this test. Plots show distributions of time required by invocations to conclude the small chunks upload step. Whiskers group the 90% of data.

Previous tests show that the delay experimented in the upload/download speed of some invocations still appears with less Lambda invocations.

The reduce phase of a MapReduce-based application requires to wait until all mappers have finished before performing the reduction in order to obtain the results. And this approach is also exhibited by typical job execution patterns that combine a loosely-

coupled phase composed by multiple parallel independent jobs with a post-processing phase that requires the aggregation of results once they are available. Therefore, it is of paramount importance to understand and characterise the performance of opaque computing platforms such as AWS Lambda.

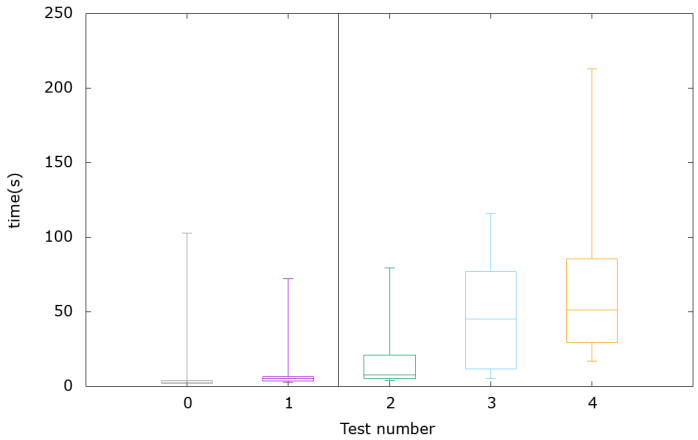

Figure 12: Network performance assessment for Lambda function with $3008MB$. Represented tests use 20 and 40 chunks in the left and right of the vertical line, respectively. The number of simultaneous invocations is, in order, 20, 100, 100, 200 and 400. Plots show the distribution of required time, by the slowest invocation, to upload the corresponding number of chunks.

Figure 12 shows a box-whiskers graph with the distributions of the slowest invocation upload time across five tests. The represented tests use 20 and 40 chunks in the left and right of the vertical line respectively. Each test has been executed 20 times and the number of simultaneously invocations is, in order, 20, 100, 100, 200 and 400. It can be seen that the maximum upload time, on average, will increase with the number of concurrent invocations uploading data, showing a possible saturation or limited network resources assignation on AWS Lambda function. Since many applications only need to upload a single results file per function invocation, we performed some tests with different number of concurrent invocations and only one chunk. The results are shown in Figure 13.

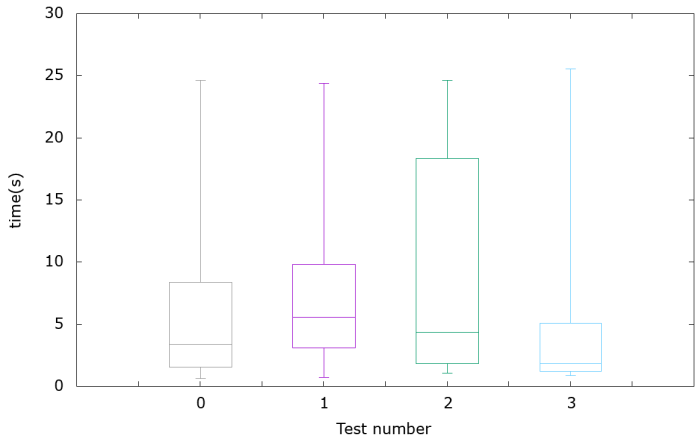

Figure 13: Network test using 100, 200, 300 and 400 Lambda simultaneous invocations with $3008MB$ and a single chunk. Plot shows distribution of required time, by the slowest invocation, to upload 1 chunk. Each test has been repeated 20 times.

With 1 chunk tests we see that the mean of the longest upload time is not correlated with the number of invoked functions. To explain this result, we have to consider two factors, the time when Lambda invocations start to upload data and the invested time to perform upload tasks. On Figure 14 we show the timestamps when invocations start to upload data on one test execution. The figure shows approximately a linear behaviour.

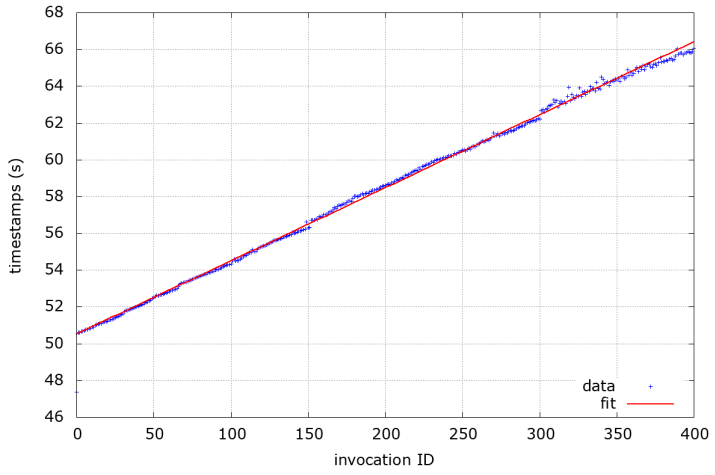

Figure 14: Timestamps from the beginning of the upload data phase of execution test using 400 Lambda invocations with 1 chunk to upload. The timestamp origin has been shifted to avoid using values on the order of 10^9. However, only time differences between Lambda invocations are relevant. The plot represent measured data (blue) and linear fit of data (red).

As expected, the linear behaviour of timestamps is repeated in all execution tests of

all tests considered in Figure 13, since the method to invoke Lambda functions is the same in all execution tests. To explain this results, we need to estimate the number of invocations that are using the network simultaneously in single executions. The mean of this value can be calculated knowing the time until another function invocation uses the network and the time used to perform the upload data:

$$SLI = \frac{UT}{slope} \tag{9}$$

where SLI is the number of simultaneous Lambda invocations using network, *slope* is the slope from linear fits and UT is the time inverted to upload data. The smaller is the *slope*, the more Lambda invocations will be using the network simultaneously. So, we use the minimum value measured of slope ($0.0498s$) to consider the worst case. Using the mean value of UT ($0.2271s$) measuring the upload times for all executions of all tests, we obtain,

$$SLI = 4.56 \tag{10}$$

This means that the first Lambda function invocation will finish its execution before invocation number 6 starts to upload data. So, on average only 5 invocations will be using the network simultaneously, no matter how many we use for the test. This result explains why the results of Figure 13 seems to not depend on the number of Lambda function invocations. However, we still occasionally measure high upload times with few Lambda functions using the network simultaneously.

4.3. Isolated network usage performance

On the previous section we identified that fluctuations were most probably caused by simultaneous network usage of several Lambda invocations. However, for completeness, some additional tests where Lambda function invocations "sequentially" use the network will be carried out in this subsection. For this tests we have used the same Lambda function and configuration than in the previous section. However, in this tests, we ensure that only one Lambda function invocation is sending data to S3, that is, we try to avoid concurrent network usage. To this aim, two kind of tests have been performed:

- **Concurrent Execution, Sequential Upload**: All Lambda invocations will be done by the invocation with ID 0 with no delay. However, each Lambda invocation will wait a time proportionally to its own ID using the Python *sleep* function. The time waited by each invocation is:

$$delay\,(seconds) = 6 \cdot (1 + ID) \tag{11}$$

  where ID is the invocation identifier that is in the interval $[0, N_{invocations}]$. The $+1$ is used to avoid sleeps of 0 seconds. Because invocation with ID 0 spends additional time invoking all other ones, an additional delay has been set to it. That delay is,

$$delay_{ID=0} = 6 \cdot (N_{invocations} + 1) \tag{12}$$

With this approach, we force the invocation with $ID = 0$ to upload its data when all others have done it. The chosen delay unit (6 seconds) has been selected after testing the time required for one function to perform the upload step with 20 chunks. The total time spent by function with $ID = 1$ with previous configuration is, on average, 14 seconds, where 12 are from sleep ($6 \cdot (1 + 1) = 12s$). Therefore, approximately, only two seconds are required to execute the entire function. Tripling this value should be enough to ensure that previous invocation has finished. Thanks to the recently timeout increment on AWS Lambda functions (to 15 minutes), we can run 100 invocations safety for that test. Notice that this test forces Lambda invocations to run concurrently but avoid network sharing.

- **Sequential Execution, Sequential Write**: In that second kind of test, only one Lambda invocation will be running at once. The invocations will be done from our local machine in loop with the flag `-invocation-type RequestResponse`. So, only when a previous invocation finishes its execution, the next one will be invoked. This approach avoids any kind of concurrence and network sharing.

In any of the previous proposed tests the expected result should be small fluctuations on upload times but not significant ones. Notice also that, in these tests, the write steps on S3 are not concurrent. Both tests have been repeated using the *Reserve concurrency* option to ensure that all invocations can be done without reaching the concurrence limit. However, this appears to have no influence on the measured results.

For the "Concurrent Execution, Sequential Upload", two tests of 10 executions each one have been carried out. The results are shown in Figure 15. The two upload steps reproduce the same behaviour, so only the “big chunk” step will be shown.

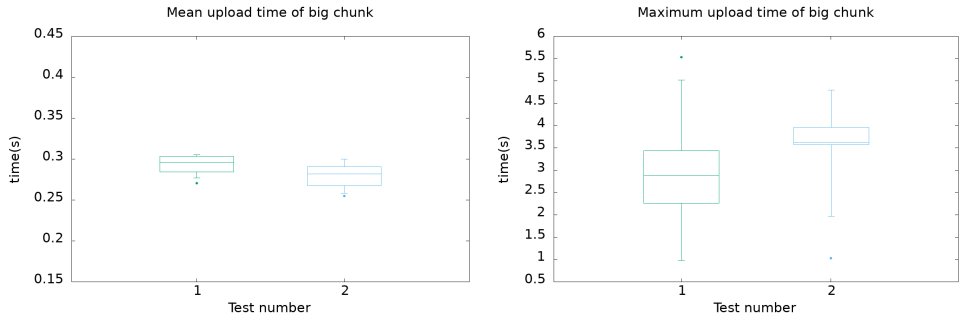

Figure 15: Distributions of mean upload time (left) and maximum upload time (right) of big chunk step for test executions of each test. Each test has performed 10 executions with 100 Lambda invocations each one and 20 chunks. Whiskers include the 90% of all data so, one test execution is shown as a point.

As we can see, both tests show the same behaviour. The mean of upload times has small fluctuations. However, few invocations require an unusually high time to perform the upload to S3. This behaviour is repeated in both big and partitioned chunk steps. Therefore, the delay apparently produced by simultaneous uploads to S3 still appears on concurrent executions with “isolated” network usage. To compare with the results of simultaneous upload tests from previous section, Figure 16 shows the results of Figure 12, but only the test corresponding to 20 chunks and 100 invocations per test execution.

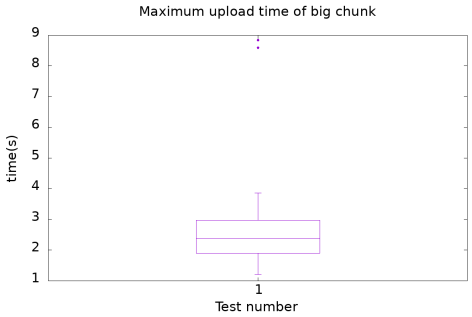

Figure 16: Results of test 1 of figure 12.

The tests performed in this section experiment a smaller delay than the one produced with simultaneous uploads. The behaviour, on average, of the slowest invocation is approximately the same in both simultaneous and isolated network usage tests. This results suggest that the delays are not produced by the number of invocations using the network but for the number of concurrent invocations. Notice that these results are perfectly compatible with the tests in Figure 13. As the invocations finish their execution before all others start, the number of concurrent invocations running is, approximately, constant. The same happens with the tests with more chunks and same number of functions or vice versa, the number of concurrent invocations is greater and the delay is greater too.

The next tests correspond to the "Sequential Execution, Sequential Write". Under these circumstances, the system cannot experiment any kind of concurrency, since all Lambda invocations will be executed sequentially. Three tests have been done with 20 chunks and 10, 100 and 100 invocations respectively and the results are shown in Figure 17.

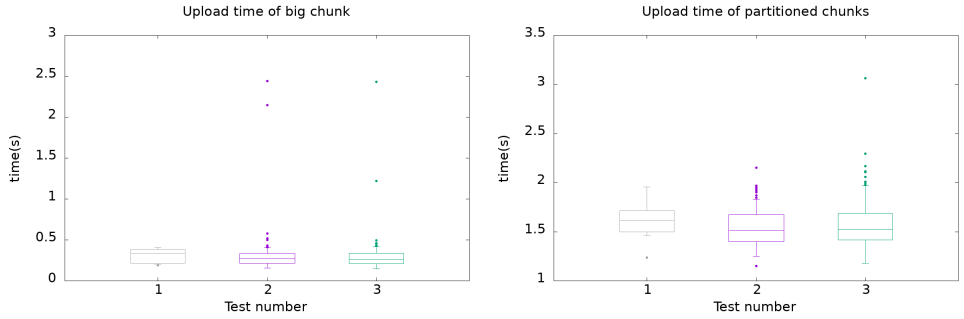

Figure 17: Upload time distributions on network tests where only one invocation is executed at once. Each test uses 20 chunks and execute 10, 100 and 100 invocations respectively.

The results indicate that still exists this effect of “slow” uploads, even when only one function is invoked. Moreover, the time required by the slowest ones are in the range obtained in previous test (Figure 15). Therefore, it seems that the invocation concurrency is not the cause of the experimented delays. The delays with no simultaneous network

usage seems to be produced by the underlying and unknown behaviour of the infrastructure provided by AWS. However, the results indicate that the invocation concurrency accentuate this effect.

4.4. Mitigation Strategies

This section identifies certain usage patterns that can be adopted by applications running on AWS Lambda in order to mitigate the heterogeneous performance results obtained as a result of the performance assessment carried out in the previous section:

- **Prepare for the worst**: As seen in previous sections, VMs with very different performance can be used by AWS to automatically execute a Lambda function invocation even if the user allocated the same amount of memory. Therefore, the user should ensure that the function invocation can run its code even when the slowest underlying VM is assigned in order to avoid time-outs.
- **Handle time-outs**: Users should assume that, eventually, a Lambda function invocation will cause a timeout provoked by a "slow network" data transfer. Therefore, the application workflow should correctly handle these errors. As an example, temporary data in S3 should not be deleted until the partial/final results have been uploaded to the corresponding permanent data storage.
- **Scalability**: According to the performance assessment carried out, applications that simultaneously upload/download data running on AWS Lambda will not scale well. However, if the application invokes Lambda functions in response to events distributed "randomly" (such as user requests) the number of functions that simultaneously use the network will be much smaller than an application where multiple functions are invoked at once, as it happens in our serverless-based MapReduce architecture. As seen before, this effect increases with the data size to transfer.
- **Avoid excessive partitioning**: Whenever possible, uploading/downloading big data chunks results in better performance that performing data transfers of a large number of small files into Amazon S3.

5. Conclusions and Future Work

This paper has introduced the MARLA (MApReduce on Lambda) framework in order to support serverless MapReduce on top of Amazon Web Services. The framework introduces additional optimisations with respect to the state of art regardless the automatic determination of mappers and the efficient usage of Amazon S3 for enhanced data transfers. When comparing the performance results with the serverless MapReduce reference architecture of AWS, we identified that AWS Lambda exhibits an heterogeneous performance behaviour that clearly has an impact on the execution time of MapReduce jobs. Therefore, single values for the execution time of an application in AWS Lambda cannot be used as a fair comparison mechanism. Instead, a careful analysis is required to provide statistical dispersion of the execution time.

To this aim, this paper undertook a thorough analysis of AWS Lambda in terms of CPU performance and network throughput with a special focus on identifying the reasons why this heterogeneity is exhibited. A correlation was obtained between the CPU model

of the underlying EC2 instances on which the Lambda functions were running and their execution time. Therefore, the claim that the performance of an AWS Lambda function is related to the allocated amount of RAM (chosen by the user) should be complemented with the fact that, depending on the underlying instance that actually executes the function (chosen by AWS), the function will exhibit execution times that can be up to 51% slower.

This poses a problem for coupled job executions, like MapReduce, in which a delay in just one the mappers affects the whole execution since the reduce phase cannot be started. This fluctuation in the execution can eventually cause timeouts in the Lambda functions and disrupt job executions if they have not been configured to accommodate this effect. This has also an economic impact since the pricing model of AWS Lambda depends on the amount of milliseconds executed, together with the RAM allocated to the Lambda function.

A similar effect was identified in the network performance. Multiple Lambda functions that put or get data in S3 buckets can produce, depending on the number of concurrent functions and size of data to store, a timeout on "slower" functions. However, with a good configuration of the application, the slow Lambda function will be re-invoked and, eventually, will finish its execution. Therefore, this effect can produce an increment in time and cost but the execution will be done eventually, as exemplified in our architecture.

Still, AWS Lambda provides an ideal computing platform in which automated scaling is handled by the Cloud provider and application developers can benefit from it for general purpose computing. Our presented architecture provides a general MapReduce platform completely executed in a serverless environment with a pay-per-use pricing model. This really stands out as a convenient approach to offer a MapReduce service that is pre-deployed as a set of Lambda functions at zero cost and the execution pipeline is triggered by uploading a file into a bucket, resulted in automated scaling to perform a MapReduce job tailored to the dataset to be processed in terms of the number of mappers.

Future works include extending the framework to support additional programming languages as well as other Cloud providers. In particular, we would like to study whether the fluctuations in AWS Lambda performance are present in the corresponding service offerings by other Cloud providers.

Acknowledgment

This study was supported by the program "Ayudas para la contratación de personal investigador en formación de carácter predoctoral, programa VALi+d" under grant number ACIF/2018/148 from the Conselleria d'Educació of the Generalitat Valenciana. The authors would also like to thank the Spanish "Ministerio de Economía, Industria y Competitividad" for the project "BigCLOE" with reference number TIN2016-79951-R.

References

[1] Amazon s3 request rate. https://docs.aws.amazon.com/AmazonS3/latest/dev/request-rate-perf-considerations.html. Accessed: 2018-12-19.

[2] Amplab benchmark. https://amplab.cs.berkeley.edu/benchmark/. Accessed: 2018-10-1.

[3] Apachehadoop. https://hadoop.apache.org/. Accessed: 2018-10-1.

[4] Aws. https://aws.amazon.com/. Accessed: 2018-10-1.

[5] Aws lambda. https://aws.amazon.com/lambda/. Accessed: 2018-10-1.

[6] Azure. https://azure.microsoft.com/. Accessed: 2018-10-1.

[7] Cloud dataproc. https://cloud.google.com/dataproc/. Accessed: 2018-10-1.

[8] Configuring lambda functions. https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html. Accessed: 2018-09-24.

[9] Corral. https://github.com/bcongdon/corral. Accessed: 2018-10-8.

[10] Emr. https://aws.amazon.com/emr/. Accessed: 2018-10-1.

[11] Gcp. https://cloud.google.com/. Accessed: 2018-10-1.

[12] Hadooptable. https://hadoopecosystemtable.github.io/. Accessed: 2018-10-1.

[13] Hdinsight. https://azure.microsoft.com/en-us/services/hdinsight/. Accessed: 2018-10-1.

[14] Lambda MapReduce, howpublished = https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/, note = Accessed: 2018-10-1.

[15] Ooso. https://github.com/d2si-oss/ooso. Accessed: 2018-10-1.

[16] Openstack. https://www.openstack.org/. Accessed: 2018-10-1.

[17] Openstack sahara. https://docs.openstack.org/sahara/latest/. Accessed: 2018-10-8.

[18] Pywren. https://github.com/pywren/pywren. Accessed: 2018-10-1.

[19] S3 multipart upload. https://docs.aws.amazon.com/AmazonS3/latest/dev/qfacts.html. Accessed: 2018-09-25.

[20] S3 scalability. https://aws.amazon.com/blogs/aws/amazon-s3-performance-tips-tricks-seattle-hiring-event/. Accessed: 2018-09-25.

[21] Wosc. https://www.serverlesscomputing.org/wosc17/. Accessed: 2018-10-1.

[22] Alex Glikson, Stefan Nastic, Schahram Dustdar. Deviceless edge computing: extending serverless computing to the edge of the network. *Conference: the 10th ACM International Systems and Storage Conference*, 2017.

[23] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, Michael Stonebraker. A comparison of approaches to large-scale data analysis. *SIGMOD 2009*, 2009.

[24] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[25] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. *Proceedings of the 2017 Symposium on Cloud Computing*, pages 445–451, 2017.

[26] Garfinkel, Simson L. An evaluation of amazon's grid computing services: Ec2, s3, and sqs. *Harvard Computer Science Group Technical Report TR-08-07*, 2007.

[27] Hyungro Lee, Kumar Satyam, Geoffrey Fox. Evaluation of production serverless computing environments. *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018.

[28] Kamil Figiela, Adam Gajek, Adam Zima, Beata Obrok, Maciej Malawski. Performance evaluation of heterogeneous cloud functions. *Concurrency and Computation*, 2018.

[29] Laing Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, Michael Swift. Peeking behind the curtains of serverless platforms. *Annual Technical Conference*, 2018.

[30] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara. Serverless computing: An investigation of factors influencing microservice performance. In *Cloud Engineering (IC2E), 2018 IEEE International Conference on*, pages 159–169. IEEE, 2018.

[31] Mengting Yan, Paul Castro, Perry Cheng, Vatche Ishakian. Building a chatbot with serverless computing. *MOTA '16 Proceedings of the 1st International Workshop on Mashups of Things and APIs*, 2016.