Improving Existing WMS for Reduced Makespan of Workflows with Lambda

Ali Al-Haboobi\textsuperscript{1(✉)} and Gabor Kecskemeti\textsuperscript{1,2(✉)}

\textsuperscript{1} Institute of Information Technology, University of Miskolc, Miskolc, Hungary \{al-haboobi,kecskemeti\}@iit.uni-miskolc.hu
\textsuperscript{2} Department of Computer Science, Liverpool John Moores University, Liverpool, UK g.kecskemeti@ljmu.ac.uk

Abstract. Scientific workflows are increasingly important for complex scientific applications. Recently, Function as a Service (FaaS) has emerged as a platform for processing non-interactive tasks. FaaS (such as AWS Lambda and Google Cloud Functions) can play an important role in processing scientific workflows. A number of works have demonstrated their ability to process these workflows. However, some issues were identified when workflows executed on cloud functions due to their limits (e.g., stateless behaviour). A major issue is the additional data transfer during the execution between object storage and the FaaS invocation environment. This leads to increased communication costs. DEWE v3 is one of the Workflow Management Systems (WMSs) that already had foundations for processing workflows with cloud functions. In this paper, we have modified the job dispatch algorithm of DEWE v3 on a function environment to reduce data dependency transfers. Our modified algorithm schedules jobs with precedence constraints to be executed in a single function invocation. Therefore, later jobs can utilise output files generated from their predecessor job in the same invocation. This reduces the makespan of workflow execution. We have evaluated the improved scheduling algorithm and the original with small- and large-scale Montage workflows. The experimental results show that our algorithm can reduce the overall makespan in contrast to the original DEWE v3 by about 10%.

Keywords: Scientific workflows · Cloud functions · Serverless architectures · Makespan

1 Introduction

A scientific workflow application consists of a large number of dependent jobs with complex precedence constrains between them, e.g. Montage [5], CyberShake [4], and LIGO [1]. These applications need large resources for processing such as cloud computing. Moreover, they require Workflow Management Systems (WMS) such as Pegasus [3] and Kepler [2] for handling the jobs. These...
WMSs help to keep the applications constraints by following a specific order of processing and ensuring the availability of data dependencies.

Function as a Service (FaaS) is a commercial cloud platform (e.g., AWS Lambda) for running distributed applications with highly scalable processing capabilities. FaaS platforms often have significant limitations on individual invocations e.g., temporary storage and memory. Several studies (e.g., [6, 13]) have investigated whether large-scale scientific workflows can be executed on function platforms in spite of their limitations. They execute scientific workflows in functions by sending a set of jobs to be run in each function invocation. When invocations complete successfully, FaaS services remove all temporary files due to their storage space limits (e.g., 500 MB in case of Lambda). Thus, output files (which can act as data dependencies for other jobs) resulting from the invocations need to be transferred to an object storage. Unfortunately, this leads to increased communication costs. As a result, the total workflow execution time (makespan) will be longer due to more dependency movement than on non-function based platforms.

[11] presented a prototype for workflows on cloud functions. In [6] the DEWE v3 is introduced that is also able to process scientific workflows using AWS Lambda and Google Cloud Functions (GCF). DEWE can process scientific workflows in three different execution modes: traditional cluster, cloud functions, and hybrid mode (combining the other two modes). Both mentioned WMS solutions were evaluated with the Montage workflow (which is a compute- and data-intensive, astronomy focused scientific application). Testing with smaller Montage workflows does not show significant differences with regards to makespan. Larger scale ones show the deficiency of these previous approaches though: their scheduling algorithm sends jobs to functions without considering job precedence requirements. Jobs with precedence could be sent as a single set for their corresponding function invocation. Thus large scale montage exhibits the problem of increased makespan due to more frequent dependency transfers.

We have changed the scheduling algorithm of DEWE v3 to reduce data dependency transfers. Our improved algorithm schedules jobs with precedence requirements to be executed in a single function invocation. It schedules a predecessor job with its successor jobs that have no other predecessor jobs to the same shard to run in the same function invocation. As a result, we moved some workflow management behaviour inside the functions, as these functions now need to assure the job ordering when they process them. Consequently, successor jobs can utilise the output files generated from their predecessor job in the same function invocation. This will ensure we don’t need to transfer the dependencies to the object store prematurely. We can schedule jobs with precedence constraints in a single shard. Because they will be captured in order by the shard and Lambda. Subsequently, Lambda will be immediately getting these jobs in a single batch.

We have evaluated our approach with AWS Lambda as our target platform. AWS offers Kinesis for queueing tasks to particular functions. Function instances have their own queues (called shards). When a function completes its current task, Lambda will pull all or some of the jobs with precedence requirements based on its batch size and the sequence of the jobs on a shard. Consequently, the remaining jobs will be waiting on the shard for the next Lambda invocation.
In our experiment, we have evaluated our improved approach and the original DEWE v3 with the 0.10° and 6.0° Montage workflows. The 6.0° workflow can be considered large-scale as it has a total of 8,586 jobs with a data dependency size of 38 GB. In addition, DEWE has already shown a large amount of re-transfer data behaviour. We used the small 0.1° workflow to show that our approach does not alter the performance. Due to Lambda’s limitations, some files cannot be processed in functions, for these we used one sufficiently sized VM.

The experimental results show that our improved system can outperform DEWE v3 in most cases. At best, in contrast to the original, we have 10% shorter makespan. This demonstrates that our improved scheduling algorithm can reduce the execution time of scientific workflows on the Lambda platform.

The remainder of this paper is structured as follows. In Sect. 2, we will present background knowledge and related works. In Sect. 3, we will explain the improved algorithm and how we changed DEWE v3. We evaluated our improvements and contrasted it to the original algorithm in Sect. 4. Finally, Sect. 5 concludes the paper and suggests some future works.

2 Background Knowledge and Related Works

2.1 Background Knowledge

A workflow can be modelled as a Directed Acyclic Graph (DAG) that consists of a set of jobs which follow a specific order in processing. The vertices represent the workflow jobs and the edges represent data dependencies between these jobs.

Executing workflows on IaaSs leads to the challenge of determining the number of VMs to back the workflow’s jobs. There are different numbers of jobs in each phase of the workflow which all could require different levels of computing resources. In order to speed up the workflow execution, we can add more VMs for processing the jobs in a certain phase. However, this may lead to a resource under-utilization issue for other phases. Unless the workflow management system is capable of dynamically reducing the number of backing VMs.

AWS presented Lambda\(^1\) in 2014 while Google introduced cloud functions (GCF\(^2\)) in 2016. The advantage of using cloud functions is the dynamic automation of resource provisioning by scaling up or down based on the workflow execution requirements. Moreover, the billing interval for cloud functions is based on 100 ms while recently Amazon and Google have changed the interval billing of virtual machines from per-hour to per-second. The function is stateless and its runtime environment is instantiated and terminated for each function invocation. Additionally, Microsoft and IBM have presented their own versions of FaaS that are Microsoft Azure Functions\(^3\) and IBM OpenWhisk Functions\(^4\).

---

1. [https://aws.amazon.com/lambda/](https://aws.amazon.com/lambda/).
2. [https://cloud.google.com/functions/](https://cloud.google.com/functions/).
3. [https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview](https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview).
4. [https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-getting-started](https://cloud.ibm.com/docs/openwhisk?topic=openwhisk-getting-started).
If workflows are executed on one of the above FaaS systems, the dynamic management of backing VMs becomes unnecessary by WMSs as FaaS systems include automated resource management in the background. Therefore, the number of concurrent invocations into the infrastructure can more closely follow the actual workflow’s demands.

[10] presented an evaluation for the following serverless computing providers: Amazon, Microsoft, Google, and IBM. They tested the function platforms by invoking CPU, memory, and disk-intensive functions. They found that at the time of writing, Amazon’s was better. In addition, they also pointed out that computing with cloud functions is more cost-effective than virtual machines due to zero delay in booting up new resources. In addition, they also pointed out that costs only get charged for the function’s actual execution time rather than paying for the idle periods as well for virtual machines. As a result, we have chosen Lambda to run workflows due to its efficiency and effectiveness comparing with other platforms.

2.2 Related Works

[11] suggested five different architectures for executing scientific workflows on the cloud. It presented a prototype for serverless computing that integrated the HyperFlow engine with GCFs and AWS Lambda. It designed to investigate the feasibility of executing compute- and data-intensive scientific workflows on cloud functions. It was tested with 0.25° and 0.4° Montage workflows and they found the approach highly promising. In addition, in [12] they presented the same prototype and tested it with the same previous montage workflows as well as a 0.6° Montage workflow. They run 0.6° as the largest workflow due to Lambda limits for the temporary storage 500 MB. Their approaches also manifest the deficiency with large-scale workflows that causes increased data dependency transfers.

[6] developed a more advanced system called DEWE v3 that is able to process scientific workflows on three different modes: traditional cluster, cloud functions, and a hybrid mode that combines the two. They evaluated it with small- and large-scale Montage workflows. The system has demonstrated the ability of cloud functions to run large-scale scientific workflows with complex precedence constraints. However, it uses a scheduling algorithm that batches jobs to Lambda without considering jobs with precedence requirements to run in a single Lambda invocation. As a result, more data dependency transfers can occur during execution between storage service and the Lambda invocation execution environment. Thus, this leads to increased communication costs.

[9] outlined the challenges for executing scientific workflows on a serverless computing platform. They proposed a Serverless Deadline-Budget Workflow Scheduling (SDBWS) algorithm that was modified to be applicable for function platforms. It was tested with the small-scale 0.25° Montage workflow on AWS Lambda. The algorithm used different memory sizes for Lambda based on the deadline and budget constraints assigned by the user. In addition, the function
resource is selected depending on the combined factors: cost and time. Their approach also exhibits the deficiency with large-scale workflows that causes increased data dependency transfers.

[13] presented work for evaluating three cloud function platforms which are Lambda, GCF, and OpenWhisk (from IBM). They evaluated the platforms with a large-scale (over 5000 jobs in parallel) bag-of-tasks style workflow. The experimental results showed that Lambda and GCF can provide more computing power if one requests more memory, while OpenWhisk is not offering an important difference. Consequently, cloud functions can provide a high level of parallelism for workflows with a large number of parallel tasks at the same time. However, they experimented with a bag-of-tasks approach where they did not consider data dependency transfers.

3 Improving an Existing Workflow Management System

DEWE v3 has three different execution modes including a cloud functions mode that we targeted to implement our approach on it. The functions mode runs workflows on FaaS platforms such as AWS Lambda. Inside this mode, we modified the AWS Lambda specific job dispatch algorithm to reduce data dependencies transfer. In the next paragraphs, we will put our modifications in context, then we will discuss our approach in detail.

The DEWE v3 system reads the workflow definition from an XML file. After the definition is loaded, the job binaries and input files are uploaded to Amazon S3. In the beginning, all jobs that have no predecessor jobs are scheduled to Amazon Kinesis stream into a common job queue. The scheduling algorithm works by predecessor jobs triggering their successor jobs when they are completed. Moreover, the Lambda function pulls a set of jobs from the Kinesis shards based on the Lambda's batch size. Therefore, the maximum amount of jobs in a single function invocation will be limited by the batch size. Depending on configuration, Kinesis can have one or more shards. Each shard acts as an independent queue for a particular function instance. Therefore, the number of parallel Lambda invocations is equal to the number of shards in Kinesis. By increasing the number of shards, DEWE v3 can speed up the execution time of scientific workflows. Figure 1 shows the original algorithm of DEWE v3 that we will compare it with our improved algorithm.

Many challenges are exposed when workflows executed on cloud functions due to different constraints. Firstly, functions have some limits such as the amount of compute, memory, and storage resources that can use them to run and store functions. For instance, Lambda needs to delete all the temporary files when a set of jobs are successfully executed due to its storage space limit 500 MB. Therefore, more data dependencies transfer occur during the execution between object storage and the function invocation environment, leading to more communication costs. Secondly, functions are stateless leading to the requirement that the output data files must be stored on an external service.
3.1 Proposed Approach

We started to implement our approach in DEWE v3 to lessen data dependency transfers and to reduce the makespan of the workflow execution. We choose DEWE v3 as the foundation because it was the closest existing open source WMS to our goals. We changed its job dispatch algorithm to improve its data dependency transfers. We moved the following workflow management behaviour inside shards and the Lambdas. Jobs with precedence constraints will be sent to the same Kinesis shard (in the order they need to be executed). Subsequently, each shard will assure the sequence of the jobs in processing by Lambda. To guarantee strictly increasing ordering, we have sent jobs serially to a shard. Additionally, we used the SequenceNumberForOrdering parameter that guarantees the order of jobs on a shard\(^5\). Next, Lambda will receive a batch of jobs based on its batch size to run them sequentially in a single invocation. But before starting any job on the Lambda, each job will read its data dependencies from the S3 storage and then storing again the output files to it. Moreover, running jobs sequentially in a single Lambda invocation will benefit our improved algorithm that it must preserve the dependence constraints. Finally, if there are remaining jobs on the shard, they will wait for the next Lambda invocation for executing.

We extended the (LambdaWorkflowScheduler) class of DEWE v3 that discussed previously. The extension called the (LambdaShardWorkflowScheduler) class where we modified the (setJobAsComplete) method to be called by the (jobCompleted) method. It schedules each predecessor job with its successor jobs that have no other predecessor jobs. Afterwards, Lambda will pull all or some of the jobs with precedence requirements based on its batch size and on their order on a shard. In addition, the remaining jobs will be waiting on the shard for the next Lambda invocation.

Algorithm 1 shows the pseudo-code of the improved algorithm for scheduling a workflow. Here we explain the steps of the improved scheduling algorithm. However, step 1 and step 6 are already in the DEWE v3 algorithm but we will explain them to show the content. Step 2: it reads the XML file of the

---

\(^5\) https://docs.aws.amazon.com/kinesis/latest/APIReference/API_PutRecord.html.
Algorithm 1. The improved scheduling algorithm.

1: procedure Scheduling jobs($T, KS$)
2:    Read the workflow definition (dag.xml)
3:    $T \leftarrow$ job, $KS_n \leftarrow$ Kinesis – shard, $Ln \leftarrow$ Lambda – function
4:    numJobs $\leftarrow$ number – of – batch, maxBatch $\leftarrow$ Lambda – batch – number
5:    flag $\leftarrow$ max – batch – number, loadBalancing[] $\leftarrow$ numberJobs – to – shard
6:    Schedule all jobs that have not predecessor jobs ($Ti$)
7:    if ($Ti$ has completed processing) then
8:        KSn=select a shard with the minimum number of receiving jobs
9:        numJobs $= 0$
10:   for each successor job of $Ti$ do
11:       $Tj \leftarrow$ successor – job – of – $Ti$
12:       Remove $Ti$ as a predecessor job from $Tj$
13:       if ($Tj$ has no other predecessor jobs) then
14:          schedule $Tj$ to $KSn$ to run on $Ln$
15:          numJobs $= numJobs + 1$
16:       while ($Tj$ has successor jobs) do
17:         if (The successor job has only one predecessor job $Tj$) then
18:             Remove $Tj$ as a predecessor job from its successor job
19:             Schedule the successor job of $Tj$ to $KSn$ to run on $Ln$
20:             numJobs $= numJobs + 1$
21:         end if
22:         if (numJobs $== maxBatch$) then
23:             flag $= true$
24:             break
25:         end if
26:       end while
27:     end if
28:     if (flag $== true$) then
29:        loadBalancing[$n$] $= loadBalancing[n] + numJob$
30:        KSn=select a shard with the minimum number of receiving jobs
31:        flag $= false$
32:        numJobs $= 0$
33:     end if
34:   end for
35:   loadBalancing[$n$] $= loadBalancing[n] + numJob$
36: end procedure

Workflow definition. Step 3: we denote $T$, $KSn$, and $Ln$ for jobs, Kinesis shards, and Lambda functions respectively. Step 4: we symbolize numJob for counting the number of batch jobs that will be sent for each shard in order to be less or equal to the number of a batch size of Lambda (maxBatch). Step 5: we denote flag as a boolean value to alert when the number of jobs that need to send them to shard be equal to the number of a batch size of Lambda. In addition, we symbolize loadBalancing as an array to count the jobs that send to each shard in order to balance the workload across Kinesis shards. Step 6: at the beginning
of the scheduling, all jobs that have not predecessor jobs will be scheduled across Kinesis shards without their successor jobs such as job 1 in step 1 of Fig. 2 and Fig. 1. In Step 7, if a predecessor job completed the execution, then in step 8: the algorithm will select a shard with the minimum number of receiving jobs among all shards. Step 10: it will traverse all successor jobs of the completed job and in step 12: each successor job will remove the completed job as a predecessor job. Next, in steps 16–21, each predecessor job will be scheduled with its successor jobs that have no other predecessor jobs. For example, job 2 schedules with its successor jobs 4 and 5 on the same Kinesis shard as illustrated in step 2 of Fig. 2. Additionally, job 3 schedules with its successor job 7 on the same Kinesis shard. But job 6 has not been scheduled with its predecessor () because it has two predecessors (jobs 2 and 3). Steps 22–25 are to stop sending jobs to shard due to reaching the maximum number of a batch size of Lambda. Steps 28–31 are to count the jobs that send to each shard. In order to balance the workload across Kinesis shards that is leading to balanced use of all Lambda instances. Moreover, step 30 is to select another Kinesis shard with the minimum number of receiving jobs among all shards to continue scheduling the remaining jobs of the workflow.

Figure 2 shows the steps of the improved scheduling algorithm while Fig. 1 illustrates the steps of the original algorithm. In Step 1, both algorithms have the same assignment to a shard. Step 2 is the difference between Fig. 2 and Fig. 1. In the original algorithm, step 2 as only jobs 2 and 3 are ready and assigned them to shards. While in the improved algorithm, we schedule all these jobs with their successor jobs (4, 5, and 7) except job 6 that has two predecessor jobs (2 and 3). Jobs will be captured in order based on our previous discussion by the shard and Lambda. Where we can put jobs with precedence constraints in a single shard. Subsequently, Lambda will be immediately getting these jobs in a single batch. When jobs 2 and 3 are completed execution in the original algorithm, they will trigger their successor jobs (4, 5, 6, and 7) in step 3 of Fig. 1. Finally, both algorithms have completed the execution of jobs in step 4.
4 Evaluation

In this section, we have evaluated the improved algorithm by comparing its results with the original algorithm. We tested them with a 0.1° and a 6.0° Montage workflows (an astronomy application). We have selected Montage because the previous studies used Montage as well and it allows easier comparison between with past results. Montage was also used for different benchmarks and performance evaluation in the past [7].

First, we tested the Montage workflow degrees by considering the data dependencies between jobs. We placed the Montage workflows on the S3 storage bucket for reading/writing by Lambda. We run the workflow management system on the virtual machine as a management node, where the VM is t2.micro instance as a free tier with 1 vCPU 2.5 GHz, Intel Xeon Family, and 1 GiB memory. In our experiment both systems were evaluated on the same platform configuration as follows:

1. The Lambda Memory sizes were: 512, 1024, 1536, 2048 and 3008 MB
2. The Lambda execution duration was 900 s.
3. The batch size of the Lambda function was 10.
4. The Kinesis shard number was 2.

Figure 3 shows the makespan of both systems. Our improvements outperform the original DEWE v3 system in most cases in these small scale experiments. While the makespan of the improved system is worse than DEWE v3 for the memory 512 MB. Because testing with smaller Montage workflows does not show significant differences with regards to makespan. Memory size will impacts the makespan of workflow execution: if the user has a large-scale workflow, sending more jobs to Lambda will need a higher memory size. CPU is allocated proportionally based on the memory size where the greater size provides more computing power. Our observations here are in alignment with [13]: the lesser memory the Lambda functions have the lesser computing capabilities they have as well.

4.1 Large-Scale Evaluation

We tested both systems with the 6° Montage workflow with data dependencies on Lambda. In this configuration, Montage is a large-scale, compute- and data-intensive scientific workflow that contains a total of 8,586 tasks, 1,444 input data files, and 22,850 intermediate files, with a total size of 38 GB. It contains short jobs such as mProjectPP, mDiffFit, and mBackground while there are six long jobs: mConcatFit, mBgModel, mAdd, mImgtbl, mShrink, and mJPEG. All the short and long jobs are executed on Lambda except the mAdd jobs which are executed on a single virtual machine. The VM is needed because the size of the input/output files for mAdd exceeds the temporary storage space offered in a single Lambda function invocation. The makespan of the large-scale workflow execution of both systems is shown in Fig. 4. In this experiment, both systems were evaluated on the same platform configuration as follows:
Fig. 3. The execution time of the both systems with a 0.1° Montage workflow with job dependencies running on different Lambda memory sizes.

Fig. 4. The execution time of both systems with a 6° Montage workflow with job dependencies running on 3 GB Lambda memory size.

1. The Lambda Memory size was 3008 MB
2. The Lambda execution duration was 900 s.
3. The batch size of the Lambda function was 20.
4. The Kinesis shard number was 30.
5. The virtual machine was t2.xlarge that executed mAdd jobs with its features: 16 GiB of memory and 4 vCPUs.

We used 2 shards for the first experiment because it used a small-scale workflow. While we used 30 shards for the second experiment because it used a large-scale workflow. In addition, we cannot increase the number of shards to indefinitely because some jobs need to execute as single-thread jobs such as mBg-Model and mAdd. Figure 4 shows the reduction in makespan for the 6° Montage workflow. Because scheduling jobs with precedence constraints to be executed
in a single function invocation shows the gains. As a result, the improved algorithm can reduce data dependency transfers to speed up the overall makespan of workflow execution as illustrated in Fig. 4.

5 Conclusion

In this paper, we have presented an improvement of the job dispatch algorithm of DEWE v3 to reduce data dependency transfers. DEWE v3 is one of the Workflow Management Systems (WMSs) that provides three different execution modes: traditional cluster, cloud functions, and hybrid mode. Our improved algorithm schedules jobs with precedence constraints to be executed in a single function invocation. Therefore, successor jobs can utilise the output files generated from their predecessor job in the same invocation. This has the potential to reduce the makespan of workflow execution. We have evaluated the improved scheduling algorithm and the original algorithm with small- and large-scale Montage workflows. The experimental results show that the improved algorithm can reduce the overall makespan in contrast to DEWE v3 by about 10%.

In future work, we will extend the improved algorithm to run on heterogeneous memory sizes of cloud functions to reduce the execution time and cost. Moreover, we will extend a Workflow Management System (WMS) tool for the DISSECT-CF [8] simulator in order to be able to simulate the execution of scientific workflows in more reproducible and controlled environments.

Acknowledgements. This work was supported in part by the Hungarian Scientific Research Fund under Grant agreement OTKA FK 131793.

References

1. Abramovici, A., et al.: LIGO: the laser interferometer gravitational-wave observatory. Science 256(5055), 325–333 (1992)
2. Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., Mock, S.: Kepler: an extensible system for design and execution of scientific workflows. In: Proceedings. 16th International Conference on Scientific and Statistical Database Management, pp. 423–424. IEEE (2004)
3. Deelman, E., et al.: Pegasus: mapping scientific workflows onto the grid. In: Dikaiakos, M.D. (ed.) AxGrids 2004. LNCS, vol. 3165, pp. 11–20. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28642-4_2
4. Graves, R., et al.: CyberShake: a physics-based seismic hazard model for southern California. Pure Appl. Geophys. 168(3–4), 367–381 (2011)
5. Jacob, J.C., et al.: Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. arXiv preprint arXiv:1005.4454 (2010)
6. Jiang, Q., Lee, Y.C., Zomaya, A.Y.: Serverless execution of scientific workflows. In: Maximilien, M., Vallecillo, A., Wang, J., Oriol, M. (eds.) ICSOC 2017. LNCS, vol. 10601, pp. 706–721. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69035-3_51
7. Juve, G., Deelman, E.: Resource provisioning options for large-scale scientific workflows. In: 2008 IEEE Fourth International Conference on eScience, pp. 608–613. IEEE (2008)

8. Kecskemeti, G.: DISSECT-CF: a simulator to foster energy-aware scheduling in infrastructure clouds. Simul. Model. Pract. Theory 58, 188–218 (2015)

9. Kijak, J., Martyna, P., Pawlik, M., Balis, B., Malawski, M.: Challenges for scheduling scientific workflows on cloud functions. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 460–467. IEEE (2018)

10. Lee, H., Satyam, K., Fox, G.: Evaluation of production serverless computing environments. In: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 442–450. IEEE (2018)

11. Malawski, M.: Towards serverless execution of scientific workflows-hyperflow case study. In: Works@ Sc, pp. 25–33 (2016)

12. Malawski, M., Gajek, A., Zima, A., Balis, B., Figiela, K.: Serverless execution of scientific workflows: experiments with hyperflow, aws lambda and google cloud functions. Future Gener. Comput. Syst. (2017)

13. Pawlik, M., Figiela, K., Malawski, M.: Performance considerations on execution of large scale workflow applications on cloud functions. arXiv preprint arXiv:1909.03555 (2019)