Serverless Execution of Scientific Workflows

Qingye Jiang[1](✉), Young Choon Lee[2], and Albert Y. Zomaya[1]

[1] The University of Sydney, Sydney, NSW 2008, Australia
qjiang@ieee.org, albert.zomaya@sydney.edu.au

[2] Macquarie University, Sydney, NSW 2109, Australia
young.lee@mq.edu.au

Abstract. In this paper, we present a serverless workflow execution system (DEWE v3[1]) with Function-as-a-Service (FaaS aka serverless computing) as the target execution environment. DEWE v3 is designed to address problems of (1) execution of large-scale scientific workflows and (2) resource underutilization. At its core is our novel *hybrid* (FaaS and dedicated/local clusters) job dispatching approach taking into account resource consumption patterns of different phases of workflow execution. In particular, the hybrid approach deals with the maximum execution duration limit, memory limit, and storage space limit. DEWE v3 significantly reduces the efforts needed to execute large-scale scientific workflow applications on public clouds. We have evaluated DEWE v3 on both AWS Lambda and Google Cloud Functions and demonstrate that FaaS offers an ideal solution for scientific workflows with complex precedence constraints. In our large-scale evaluations, the hybrid execution model surpasses the performance of the traditional cluster execution model with significantly less execution cost.

Keywords: Scientific workflow · Function-as-a-service · Serverless computing

1 Introduction

Scientists in different fields such as high energy physics and astronomy are developing large-scale applications in the form of workflows with many precedence-constrained jobs, e.g., Montage [10], LIGO [1], and CyberShake [9]. Such scientific workflows often become very complex in terms of the number of jobs, the number and size of the input and output data, as well as the precedence constraints between different jobs. Typically, scientists use a workflow management system, such as Pegasus [6], Kepler [3] and Polyphony [18] to manage the execution of their workflows. This requires scientists to setup and configure clusters as the target execution environment, where the smallest unit of computing resource is either a physical server or a virtual machine. As the size of the

---

[1] DEWE v3 is the third *generation* of our Distributed Elastic Workflow Execution system for FaaS in public clouds. DEWE v3 only shares the name with the previous two versions ([11,14]), i.e., it is a complete rewriting.

© Springer International Publishing AG 2017

M. Maximilien et al. (Eds.): ICSOC 2017, LNCS 10601, pp. 706–721, 2017.
https://doi.org/10.1007/978-3-319-69035-3_51

workflow grows, setting up and configuring a large-scale cluster often becomes a challenging task, especially for researchers outside the field of high performance computing (HPC). Also, it is common to observe serious resource underutilization in large-scale clusters, primarily due to the complex precedence constraints among the various jobs in the workflow. Researchers have always been looking for new ways to (a) make it easier for researchers to execute large-scale workflows; and (b) mitigate the resource underutilization issue.

In recent years, Function-as-a-Service (FaaS) such as AWS Lambda [4] and Google Cloud Functions [8] started to gain attention in public clouds. FaaS offers compute services that run code in response to events. The computing resource is automatically managed by the public cloud service provider. The customer pays for the actual amount of computing resource consumed. The dynamic resource allocation mechanism and fine-grained pricing model seem to offer a potential solution for the above-mentioned problems. However, it remains questionable whether such transient execution environment with stringent resource limits is capable of executing large-scale workflows with complex precedence constraints.

In this paper, we present DEWE v3[1], a workflow management and execution system designed with FaaS as the target execution environment. The specific contributions of this paper are:

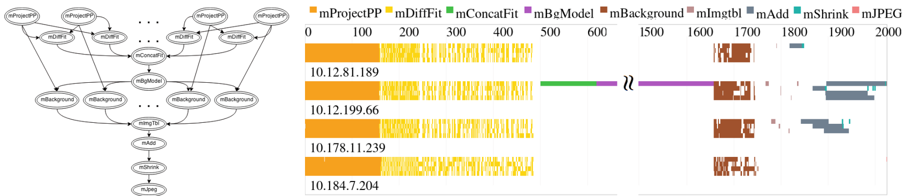
- We demonstrate that FaaS offers an ideal execution environment for scientific workflows with its dynamic resource allocation mechanism and find-grained pricing model.
- We propose and validate a hybrid execution model that is effective in dealing with the maximum execution duration limit, memory limit, and storage space limit in the FaaS execution environment.
- We demonstrate that DEWE v3 on AWS Lambda is capable of executing large-scale data-intensive scientific workflows. In our large-scale tests, the hybrid execution model achieves shorter execution time with only 70% of the execution cost, as compared with the traditional cluster execution model.
- DEWE v3 significantly simplifies the effort needed to execute large-scale scientific workflows on public clouds.

We evaluate the performance of DEWE v3 on both AWS Lambda and Google Cloud Functions with Montage scientific workflows[2]. The hybrid execution enabled by DEWE v3 takes advantage of fine-grained pricing of FaaS and efficient resource utilization of local clusters. The performance gain from the hybrid execution becomes more apparent as workflows become larger scale.

The rest of this paper is organized as follows. Section 2 describes the motivation of this work. Section 3 describes the design and implementation of DEWE v3. In Sect. 4, we evaluate the performance of DEWE v3 on both AWS Lambda and Google Cloud Functions, using a set of Montage workflows as test cases. Section 5 reviews related work, followed by our conclusions in Sect. 6.

[1] The source code is available from https://github.com/qyjohn/DEWE.v3.

[2] Montage (http://montage.ipac.caltech.edu/) is an astronomical image mosaic engine that stitches sky images dealing with hundreds or even thousands of dependent jobs. [10].



(a) Montage workflow.

(b) Detailed visualization of a 6.0-degree Montage workflow running on 4 m1.xlarge EC2 instances using DEWE v1.

Fig. 1. Execution of Montage Workflow.

2 Motivation

A workflow can be represented by a directed acyclic graph (DAG), where the vertices represent the tasks and the edges represent the precedence constraints. Figure 1a describes the structure of a Montage workflow. As the size and complexity of a workflow increases, managing its execution on a cluster with multiple nodes becomes a complex issue.

Most existing workflow management systems use clusters as the target execution environment. A cluster consists of a set of computing resources called worker nodes, where a worker node can be either a physical server or a virtual machine. To execute a workflow, scientists often need to perform a set of administrative tasks including (a) provisioning the computing resources needed; (b) setting up a cluster with an appropriate shared file system; (c) deploying the workflow management system on a master node and the job execution agent on the worker nodes; (d) monitoring the health status of all worker nodes; and (e) de-provisioning the computing resources when the work is done. These administrative tasks can be quite difficult for scientists without dedicated hardware and support staff. It is common that people with different levels of expertise come up with clusters with significant performance differences with the same set of hardware.

Because of the precedence constraints in a workflow, in certain phases during the execution only a small number of jobs are eligible to run. The traditional cluster approach presents a classical dilemma in workflow scheduling and execution – adding more computing resources to the execution environment can speed up the execution of certain phases of a workflow, but also results in significant resource underutilization during other phases of the same workflow. In recent years, researchers have attempted to address the resource underutilization issue by taking advantage of the elasticity of public clouds. This is achieved by dynamically adding worker nodes to – or removing worker nodes from – the execution environment base on the actual workload. However, such practice often results in higher costs because of the one-hour minimum charge pricing model commonly practiced by most public cloud service providers. Figure 1b visualizes the execution of a 6.0-degree Montage workflow running on a cluster with 4 m1.xlarge EC2 instances using the DEWE v1 workflow management system. The progress

of the Montage workflow has a four-stage pattern. During the second stage only two single-thread jobs `mConcatFit` and `mBgModel` are running one after another. It took 2025 s to complete the execution of the workflow, with the total cost being 4 instance-hours. If we remove 3 worker nodes after the first phase, then add 3 worker nodes back for the third phase, then the total cost would become 7 instance-hours. As such, dynamically changing the number of worker nodes in the workflow execution environment is not economically feasible without a finer-grained pricing model.

The scientific computing community has long been searching for a workflow management system that is easy to setup and use. Ideally, scientists do not need to know any details about the underlying computing resource such as worker node and file system. The amount of computing resource available in the workflow execution environment can be easily reconfigured. The execution cost should be the actual amount of computing resource consumed, not including the amount of computing resource that is wasted. However, this can not be easily achieved when the smallest unit of computing resource is a physical server or a virtual machine with an hourly pricing model.

The emergence of FaaS in public clouds provides a potential solution to the above-mentioned problem. AWS introduced Lambda in 2014 and Google introduced Cloud Functions in 2016. With FaaS, computing resource is automatically provisioned by the service provider when the function is invoked, and de-provisioned when the function finishes execution. Since the customer does not have access to the execution environment running the code, FaaS is often referred to as “serverless computing”. The customer pays for the actual amount of computing resource consumed, which is represented by the size of the function invocation environment times the duration of the invocation.

In light of the recent advancements in FaaS we develop DEWE v3, a workflow management and execution system with FaaS as the target execution environment. With DEWE v3, scientists only need to provision a single server to run the workflow management system. The jobs in the workflow are executed by the FaaS function, whose computing resource is automatically provisioned and de-provisioned by the service provider on demand. DEWE v3 uses object storage service for data staging, including workflow definition, binaries, input and output files. Researchers do not need to setup and configure a shared file system that can be accessed from all worker nodes.

3 Design and Implementation

The DEWE v3 system (Fig. 2) consists of three major components: the workflow management system, the FaaS job handler, and the local job handler. The system utilizes object storage service for binary and data staging. Different components in the system communicate with each other using a set of queues.

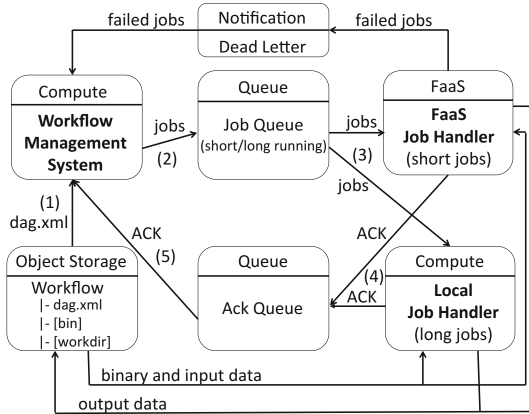


Fig. 2. The architecture of DEWE v3.

3.1 Workflow Management System

The workflow management system runs on a server, which we call the management node. The management node can be an EC2 instance, a GCE instance, or a traditional server or virtual machine. The workflow management system reads (1) the workflow definition (dag.xml) from object storage, parses the workflow definition and stores job dependencies information into a data structure. If a job has no pending dependency precedence requirements, the job is eligible to run and is published to a job queue (2), from which it will be picked up by a job handler for execution (3). When a job is successfully executed by a job handler, the job handler sends an acknowledgement message to an ACK queue (4), indicating the job is now completed. The workflow management system polls the ACK queue for completed jobs (5) and updates the status of all pending jobs that depend on the completed jobs.

The FaaS execution environment usually has a maximum execution duration limit for each invocation. The maximum execution duration limit for AWS Lambda is 300 s. The maximum execution duration limit for Google Cloud Functions is 540 s. In DEWE v3, we can define a set of long-running jobs (long.xml) for the workflow. The execution time of a particular job can be estimated from module testing or previous experiences, or derived based on the time and space complexity of the algorithm. If a job is expected to finish execution within the maximum execution duration limit, it is published into a common job queue, otherwise it is published into a specific job queue for long-running jobs.

3.2 FaaS Job Handler

The FaaS job handler is a function deployed to the respective FaaS service. For both AWS Lambda and Google Cloud Functions, the deployment process includes only three simple steps in the web console: (a) uploading the function

package to object storage; (b) specifying the name and method to run; and (c) specifying the memory footprint and default execution timeout for the function. DEWE v3 automatically creates the other components (such as the queues) needed at start up, and terminates these components at shut down.

The FaaS job handler is invoked by incoming messages in the common job queue. Each message represents a job that is eligible to run. By design, an AWS Lambda invocation can contain one or more jobs, while a Google Cloud Functions invocation contains only one job. The FaaS job handler parses the job definitions for the names of the binary and input/output files, as well as the command line arguments. It downloads the binaries and input files from object storage into a temporary folder, then executes the jobs in the temporary folder. When the jobs are successfully executed, the FaaS job handler uploads the output files back to object storage. For both AWS Lambda and Google Cloud Functions, the FaaS execution environment has only 500 MB storage space. Because of this limit, the FaaS job handler deletes all the temporary files when a batch of jobs are successfully executed. A job might fail to execute in the FaaS execution environment for various reasons, including out-of-memory error, out-of-disk-space error, or maximum execution time limit exceeded. The FaaS job handler has a fail over mechanism. If a particular job fails to execute in the FaaS execution environment, it is sent to a dead letter queue for the workflow management system to pick up. The workflow management system resubmits the job to the long-running job queue, from which it is picked up by the local job handler for execution.

Because the FaaS job handler deletes all temporary files, duplicated data transfer between object storage and the FaaS execution environment might occur during the execution, introducing additional communication cost. For example, a 2.00-degree Montage workflow contains 300 `mProjectPP` jobs, 836 `mDiffFit` jobs, and 300 `mBackground` jobs. The sizes of the `mProjectPP`, `mDiffFit` and `mBackground` binaries are 3.2 MB, 0.4 MB and 3.2 MB respectively. If the required binaries have to be transferred once for each and every job, then the binaries alone would create approximately 2 GB inbound data transfer from object storage to the FaaS execution environment. For bigger workflows with a larger number of similar jobs, such duplicated data transfer can become a serious issue.

The FaaS job handler implements two levels of caching for binaries and input/output data. The first level is 'transient' caching, which applies to multiple jobs within the same invocation in AWS Lambda. With transient caching, the FaaS job handler caches the binaries and input/output data within the same invocation, but deletes them at the end of the invocation. If in an invocation the FaaS job handler receives 10 `mProjectPP` jobs then the `mProjectPP` binary only needs to be downloaded once, reducing 90% of the repeated data transfer for the `mProjectPP` binary. The second level is 'persistent' caching, which applies to multiple invocations with the same FaaS execution environment. Both AWS Lambda and Google Cloud Functions reuse the underlying execution environments for performance considerations. If during an invocation a file is created

under the `/tmp` folder, the same file is accessible in other invocations when the execution environment is reused. However, neither AWS nor GCP (Google Cloud Platform) discloses how the FaaS execution environment is reused, so the availability of files created in previous invocations becomes non-deterministic. With persistent caching, the FaaS job handler only caches the binaries for future invocations, because the accumulated size of the input/output data is usually bigger than the amount of storage available. When the FaaS job handler is invoked, it first checks the /tmp folder for previously cached binaries, and transfers only the missing binaries for the invocation. Such persistent caching approach is inconsistent with the stateless design principle. In DEWE v3 this is an optional feature that can be turned on or off.

3.3 Local Job Handler

The local job handler is a multi-thread application running on one or more worker nodes. The level of concurrency equals the number of CPU cores available on the worker node. The local job handler polls the long-running job queue for jobs to execute. When a job is received from the queue, the local job handler parses the job definition for the name of the binary and input/output files, as well as the command line arguments. It downloads the binary and input files from object storage into a temporary folder, then executes the job in the temporary folder. When the job is successfully executed, the job handler uploads the output files back to object storage. Because the worker node usually has sufficient storage space, a caching mechanism is implemented to cache all the binaries and input/output files to avoid duplicated data transfer.

DEWE v3 has an optional switch to enforce local execution. When local execution is enforced, all the jobs in the workflow are submitted to the long-running job queue, from which they are picked up by the local job handler for execution. In this case, DEWE v3 is said to be running in traditional cluster mode.

3.4 Others

DEWE v3 is capable of running in three different modes: (a) traditional cluster mode where all jobs are executed by the local job handler running on a cluster; (b) serverless mode where all jobs are executed by the FaaS job handler running in the FaaS execution environment; and (c) hybrid mode where the short jobs are executed by the FaaS job handler, while the long-running jobs are executed by the local job handler.

On the management node we run an instance of the local job handler by default. With this hybrid approach, DEWE v3 is capable of handling both short and long running jobs, regardless of the maximum execution duration limit imposed by the FaaS execution environment, without the need to provision additional computing resource. To fully utilize the computing resource on

Table 1. The small-scale Montage workflows used in the initial evaluation.

	0.25 Degree	0.50 Degree	1.00 Degree	2.00 Degree
Jobs: mProjectPP	12	32	84	300
Jobs: mDiffFit	21	73	213	836
Jobs: mConcatFit	1	1	1	1
Jobs: mBgModel	1	1	1	1
Jobs: mBackground	12	32	84	300
Jobs: mImgtbl	1	1	1	1
Jobs: mAdd	1	1	1	1
Jobs: mShrink	1	1	1	1
Jobs: mJPEG	1	1	1	1
Input file count	17	37	89	305
Input file size (MB)	25	65	170	630
Output file count	117	353	981	3,713
Output file size (MB)	248	632	1,694	6,069

the management node, DEWE v3 provides the option to route a certain percentage of the short jobs to the long-running job queue, forcing the workflow to be executed in hybrid mode.

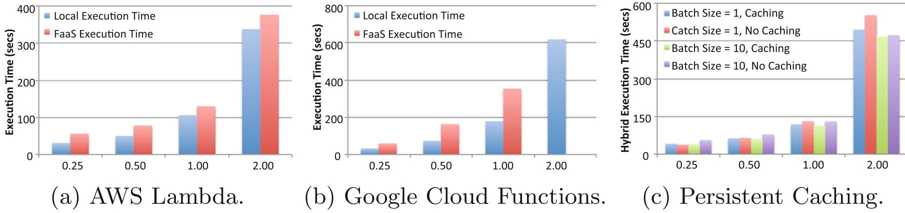
4 Evaluation

In this section, we evaluate the performance of DEWE v3 on both AWS Lambda and Google Cloud Functions. The evaluation is divided into three parts – initial evaluation, performance tuning strategy, and large-scale evaluation. For all the experiments described in this section, we perform the same experiment three times, and report the average number as the test result.

While DEWE v3 is applicable to other workflow applications, our evaluation in this study is conducted using Montage workflows due to: (1) the Montage source code and data is publicly available, (2) the project is well maintained and documented so that researchers can easily run the Montage workflow with various tools, and (3) Montage is widely used by the workflow research community as a benchmark tool to compare the performance of different workflow scheduling algorithms and workflow management systems [2,12,13,17].

4.1 Initial Evaluation

In this evaluation, we use four small-scale Montage workflows as test cases – a 0.25-degree Montage workflow, a 0.50-degree Montage workflow, a 1.00-degree Montage workflow, and a 2.00-degree Montage workflow. Table 1 lists the characteristics of these small-scale Montage workflows.



(a) AWS Lambda. (b) Google Cloud Functions. (c) Persistent Caching.

Fig. 3. Small-scale Montage workflows running on AWS and GCP with respect to different data sizes, 0.25, 0.50, 1.00 and 2.00, respectively.

With AWS, the management node is a c3.xlarge EC2 instance in the us-east-1 region. The EC2 instance has 4 vCPU, 7.5 GB memory and 100 GB general-purpose SSD EBS volume. The common job queue is a Kinesis stream with 10 shards, and the batch size of the Lambda function trigger is set to 10. The Lambda execution environment has 1536 MB memory. With GCP, the management node is a customized n1-highcpu-4 GCE instance in the us-central1 region. The GCE instance also has 4 vCPU, 7.5 GB memory and 100 GB SSD persistent storage. The Google Cloud Functions execution environment has 2048 MB memory.

In this evaluation, we carry out three sets of experiments. The first set of experiments are run in serverless mode. The only exception is the `mImgtbl` and `mAdd` jobs in the 2.00-degree Montage workflow are executed by the local job handler, because the size of the input/output files exceeds the storage space available in the FaaS execution environment. For this particular test, the 2.00-degree Montage workflow is executed in hybrid mode. The second set of experiments are run in traditional cluster mode, where all jobs are executed by the local job handler running on the management node. The third set of experiments are run in hybrid mode to evaluate the effect of persistent caching, with the `mConcatFit`, `mBgModel`, `mAdd`, `mShrink` and `mJPEG` jobs being executed by the local job handler running on the management node. In serverless mode, the execution time is noted as FaaS execution time. In cluster mode, the execution time is noted as local execution time. In hybrid mode, the execution time is noted as hybrid execution time. We do not compare the test results obtained from AWS and GCP. Instead, we focus on comparing the execution time observed on the same cloud.

The local and FaaS execution time obtained from AWS is presented in Fig. 3a. In all four test cases, FaaS execution time is slightly longer than local execution time. For the 0.25-degree workflow, FaaS execution time is 80% greater than local execution time. For the 0.50-degree workflow, FaaS execution time is 56% greater than local execution time. For the 1.00-degree workflow, FaaS execution time is 23% greater than local execution time. For the 2.00-degree workflow, FaaS execution time is 11% greater than local execution time. The FaaS execution environment has less vCPU and memory resource than the local execution environment. The local job handler caches all binaries and input/output files

throughout the execution, while the FaaS job handler downloads them for each invocation. It is expected that it takes longer for the same job to run by the FaaS job handler. When the workflow is small, the concurrent execution of a small number of jobs by the FaaS job handler is not sufficient to compensate for the above-mentioned performance lost, resulting in relatively longer FaaS execution time. As the size of the workflow grows, the concurrent execution of a larger number of jobs by the FaaS job handler gradually offset the above-mentioned performance lost, reducing the difference between FaaS execution time and local execution time. Considering the small difference between FaaS and local execution times for the 2.00-degree workflow, AWS Lambda seems to be a promising execution environment for workflows with a high level of concurrency.

The local and FaaS execution time obtained from GCP is presented in Fig. 3b. For the 0.25-degree workflow, FaaS execution time is 84% greater than local execution time. For the 0.50-degree workflow, FaaS execution time is 123% greater than local execution time. For the 1.00-degree workflow, FaaS execution time is 99% greater than local execution time. The 2.00-degree workflow fails to execute on Google Cloud Functions within a reasonable time frame due to a large number of "quota exceeded" errors. Google Cloud Functions has a default 1 GB per 100 s quota for inbound and outbound socket data transfer. Montage is a data-intensive workflows, the large amount of data transfer quickly consumes the above-mentioned quota, resulting in the "quota exceeded" errors. When this occurs, Google Cloud Functions waits for the next quota period to execute the jobs waiting in the queue, causing the extra increase in FaaS execution time. In our evaluations we are given a significant quota increase from Google, allowing us to achieve 10 GB inbound and outbound socket data transfer per 100 s. With this new limit, we still frequently encounter the same error for the 2.00-degree Montage workflow. As such, we carry out our subsequent evaluations on AWS only.

The effect of persistent caching is presented in Fig. 3c. When the batch size is 1, the effect of caching is not obvious for smaller workflows (0.25-degree and 0.50-degree), but becomes significant for bigger workflows (1.00-degree and 2.00-degree). This is because the FaaS job handler executes only 1 job during each invocation. The transient caching mechanism is not in effect, and persistent caching becomes the only optimization for binary and data staging. When the batch size is 10, the effect of persistent caching is obvious for smaller workflows (0.25-degree, 0.50-degree and 1.00-degree), but becomes insignificant for bigger workflows (2.00-degree). This is because the FaaS job handler now executes 10 jobs during each invocation. The transient caching mechanism already eliminates 90% of the duplicated transfer for the binaries, with very little space left for further optimization with persistent caching. Therefore, for the subsequent experiments reported in this paper, we turn off the persistent caching option.

4.2 Performance Tuning

In this evaluation, we use a 4.00-degree Montage workflow as the test case. The workflow has 802 `mProjectPP` jobs, 2,316 `mDiffFit` jobs, and 802 `mBackground`

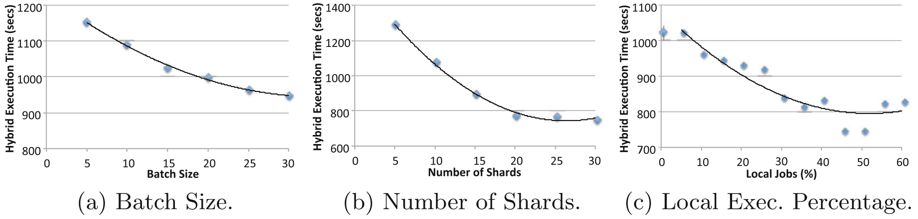
jobs, making it an ideal use case for parallel optimization. The workflow has 817 input files with a total size of 2,291 MB, and 10,172 output files with a total size of 17,010 MB. We execute the 4.00-degree Montage workflow in hybrid mode, with the `mConcatFit`, `mBgModel`, `mAdd`, `mShrink` and `mJPEG` jobs being executed by the local job handler running on the management node. These jobs are not capable of running in the Lambda execution environment because they run longer than the maximum execution duration limit, or they require more storage or memory resource than what is available. To establish a baseline for performance tuning, we execute the workflow in traditional cluster mode on the management node. The local execution time observed is 950 s.

In hybrid mode, there are three parameters that can affect hybrid execution time, including (a) the number of shards in the Kinesis stream, (b) the batch size for each invocation, and (c) the percentage of short jobs that are handled by the local job handler. In this evaluation, we carry out three sets of experiments, including (a) a fixed number of shards, all short jobs are executed by the FaaS job handler, with the variable being the batch sizes; (b) a fixed batch size, all short jobs are executed by the FaaS job handler, with the variable being the number of shards; and (c) a fixed number of shards and a fixed batch size, with the variable being the percentage of short jobs executed by the local job handler.

In test (a), we used a Kinesis stream with 10 shards as the common job queue, then change the batch size of the Lambda function trigger. As shown in Fig. 4a, the hybrid execution time decreases when the batch size increases. With transient caching, the FaaS job handler caches the binaries and input/output files needed for a particular invocation. Increasing the batch size reduces the number of invocations and the amount of duplicated data transfer, hence the decrease in hybrid execution time. However, the batch size can not be increased indefinitely, because the size of the files to be cached gradually exceeds the storage space limit. For the Montage workflow, We observe that the maximum batch size we can achieve is 30. When the batch size is bigger, we frequently observe jobs fail due to “no space left on device” errors.

In test (b), we set the batch size of the Lambda function trigger to 10, then use Kinesis streams with different number of shards as the common job queue. As shown in Fig. 4b, the hybrid execution time decreases when the number of shards increases. With AWS Lambda, the number of concurrent invocations equals to the number of shards in the Kinesis stream. Increasing the number of shards increases the number of concurrent invocations, hence the decrease in hybrid execution time. As the number of shards continues to increase, the hybrid execution time gradually converges. This is because the workflow has a set of `mConcatFit`, `mBgModel`, `mAdd`, `mShrink` and `mJPEG` single-thread jobs that run in a sequential manner. The `mBgModel` job alone takes approximately 350 s to run, accounting for approximately 45% of the hybrid execution time. These jobs now become the dominating factor in the hybrid execution time.

In test (c), we use a Kinesis stream with 10 shards as the common job queue, the batch size of the Lambda function trigger is set to 30. In addition to the long-running jobs such as `mConcatFit`, `mBgModel`, `mAdd`, `mShrink` and `mJPEG`,



(a) Batch Size. (b) Number of Shards. (c) Local Exec. Percentage.

Fig. 4. Execution time of a 4.00-degree Montage workflow on AWS.

Table 2. Large-scale test environments. Hybrid environments differ by the numbers of shards, 28 and 56, respectively; hence Hybrid-28 and Hybrid-56.

	Cluster-1	Cluster-2	Cluster-3	Hybrid-28	Hybrid-56
Instance type	c3.2xlarge	c3.2xlarge	c3.2xlarge	c3.2xlarge	c3.2xlarge
Number of nodes	1	2	3	1	1
Total vCPU cores	8	16	24	8	8
Total memory (GB)	15	30	45	15	15
Total storage (GB)	500	1000	1500	500	500
Job stream shards	0	0	0	28	56
Hourly price (USD)	0.42	0.84	1.26	0.84	1.26
Lambda function (USD)	-	-	-	0.06	0.06
Total cost (USD)	0.42	0.84	1.26	0.90	1.32

we schedule a fraction of the short jobs to the local job handler running on the management node. As shown in Fig. 4c, the hybrid execution model effectively utilize the idling computing resource on the management node, resulting in the decrease in hybrid execution time. However, when the amount of jobs routed to the local job handler exceeds the capacity of the management node, the hybrid execution time starts to increase again.

4.3 Large-Scale Evaluation

In this evaluation, we use a 8.00-degree Montage workflow with a total of 13,274 jobs as the test case. The workflow has 2,655 `mProjectPP` jobs, 7,911 `mDiffFit` jobs, and 2,655 `mBackground` jobs. The workflow has 4,348 input files with a total size of 8,524 MB, and 32,753 output files with a total size of 58,561 MB.

Traditionally, when scientists need to speed up the execution of a workflow, they add worker nodes to the cluster. With the hybrid execution model, we simply use a Kinesis stream with more shards to increase the number of concurrent invocations. To compare the performance between the traditional cluster execution model and the proposed hybrid execution model, we use the local execution time of the workflow on the management node as the baseline. The management node is a c3.2xlarge EC2 instance in the us-east-1 region, with 8 vCPU cores,

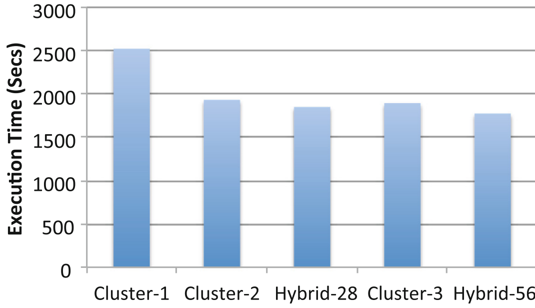


Fig. 5. Execution time of a 8.00-degree Montage workflow on AWS.

15 GB memory and 500 GB general-purpose SSD EBS volume. Then we run two sets of experiments with the same workflow. In the first set of experiments, we compare (a) the cluster execution time on a two-node cluster with 2 × c3.2xlarge EC2 instances with (b) the hybrid execution time on 1 × c3.2xlarge management node with *28 shards* in the Kinesis stream, where 20% of the short jobs are executed by the local job handler. In the second set of experiments, we compare (a) the cluster execution time on a three-node cluster with 3 × c3.2xlarge EC2 instances with (b) the hybrid execution time on 1 × c3.2xlarge management node with *56 shards* in the Kinesis stream, where 15% of the short jobs are executed by the local job handler. For both sets of experiments, the hourly cost of both execution environments is the same. In this test, the execution cost of the Lambda function falls within the AWS Lambda free-tier offering. We estimate the execution cost of the Lambda function based on the number and duration of invocations obtained from CloudWatch and multiply them with the standard pricing. The details of these test environments are listed in Table 2.

Figure 5 presents the test results. In the first set of experiments, the traditional cluster execution model (Cluster-2) achieves 18% speed-up while the new hybrid execution model (Hybrid-28) achieves 22% speed-up, as compared with the baseline obtained on Cluster-1. In the second set of experiments, the traditional cluster execution model (Cluster-3) achieves 20% speed-up while the new hybrid execution model (Hybrid-56) achieves 25% speed-up, as compared with the baseline obtained on Cluster-1. Note that Hybrid-28 achieves more speed-up than Cluster-3, while the total cost of Hybrid-28 is only 70% of Cluster-3.

5 Related Work

There have been an abundance of literature on workflow management systems such as DAGMan [5], Pegasus [6] and Kepler [3]. These frameworks use clusters with multiple worker nodes for as the execution environment. Such approaches tend to be heavy-weight and are inaccessible to scientists who lack dedicated hardware and support staff.

Polyphony [18] was designed and developed with AWS as the target execution environment, but the software is not accessible to the workflow researcher community. The work in [15] deals with scheduling scientific workflows across multiple geographically distributed resource sites; however, the scale of workflows is still limited to small, e.g., 255 tasks per workflow. All of the above-mentioned workflow management systems exhibit inefficiency in scheduling a large number of short-life jobs across multiple worker nodes.

To execute large scale scientific workflows in a cost effective way, the computing resources needed must be carefully planned. Such planning usually involves cost and performance trade-off for scientists. In recent years, researchers spend a significant amount of effort on scheduling and resource allocation algorithms to meet certain deadline and cost constraints [7,11–13,15,17,19]. These works are rather complementary and/or supplementary that can significantly benefit from using DEWE v3.

AWS introduced Lambda [4] in 2014, while Google introduced Cloud Functions [8] in 2016. Malawski [16] reviewed the various options of executing scientific workflows in serverless infrastructures. The author created a prototype workflow executor function using Google Cloud Functions, with Google Cloud Storage for data and binary storage. The author used 0.25-degree and 0.4-degree Montage workflows to evaluate the prototype and found the approach highly promising. Unlike the test cases in our study (up to 8.00-degree Montage workflow), the evaluation in [16] is limited to small-scale workflows. Also, the work in [16] failed to notice the impact of the limited inbound and outbound socket data quota on the execution of data-intensive scientific workflows.

6 Conclusion

In this paper, we present DEWE v3, a workflow management system with FaaS as the target execution environment. We present the design and implementation of DEWE v3, as well as its capability in executing large-scale scientific workflows. We demonstrate that AWS Lambda offers an ideal execution environment for scientific workflow applications with complex precedence constraints. Google Cloud Functions, in its current form, is not suitable for executing scientific workflow applications due to its limited inbound and outbound socket data quota.

We propose and validate a hybrid execution model that is effective in dealing with the various limits imposed by the FaaS execution environment. We take advantage of the hybrid execution model to speed up the workflow execution by fully utilizing the computing resource available on the management node. The largest scale experiment presented in this paper is an 8.00-degree Montage workflow with over 13,000 jobs and more than 65 GB input/output data. The hybrid execution mode achieves shorter execution time with only 70% of the execution cost, as compared to the traditional cluster execution mode. Since each Lambda function invocation can handle up to 30 jobs in one batch, further speed-up can be achieved by scheduling jobs with precedence requirements into a the same invocation. This will be addressed in our future works.

DEWE v3 reduces the effort needed to execute large-scale scientific workflows. It liberates scientist from the tedious administrative tasks involved in the traditional cluster approach, allowing them to focus on their own research work.

References

1. Abramovici, A., Althouse, W.E., et al.: LIGO: the laser interferometer gravitational-wave observatory. Science **256**(5055), 325–333 (1992)
2. Abrishami, S., Naghibzadeh, M., Epema, D.H.: Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds. Future Gener. Comput. Syst. **29**(1), 158–169 (2013)
3. 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 of 2004 16th International Conference on Scientific and Statistical Database Management, pp. 423–424 (2004)
4. Amazon Web Services: AWS Lambda (2014), https://aws.amazon.com/lambda/
5. Couvares, P., Kosar, T., Roy, A., Weber, J., Wenger, K.: Workflow management in condor. In: Workflows for e-Science, pp. 357–375 (2007)
6. Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Patil, S., Su, M.-H., Vahi, K., Livny, M.: Pegasus: mapping scientific workflows onto the grid. In: Dikaiakos, M.D. (ed.) AxGrids 2004. LNCS, vol. 3165, pp. 11–20. Springer, Heidelberg (2004). doi:10.1007/978-3-540-28642-4_2
7. Duan, R., Prodan, R., Fahringer, T.: Performance and cost optimization for multiple large-scale grid workflow applications. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, p. 12. ACM (2007)
8. GCP: Google Cloud Functions (2016), https://cloud.google.com/functions/
9. Graves, R., Jordan, T.H., et al.: Cybershake: a physics-based seismic hazard model for Southern California. Pure. appl. Geophys. **168**(3–4), 367–381 (2010)
10. Jacob, J.C., Katz, D.S., et al.: Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. Int. J. Comput. Sci. Eng. **4**(2), 73–87 (2009)
11. Jiang, Q., Lee, Y.C., Zomaya, A.Y.: Executing large scale scientific workflow ensembles in public clouds. In: Proceedings of 2015 44th IEEE International Conference on Parallel Processing (ICPP), pp. 520–529. IEEE (2015)
12. Juve, G., Deelman, E.: Resource provisioning options for large-scale scientific workflows. In: Proceedings of 2008 4th IEEE International Conference on eScience, pp. 608–613. IEEE (2008)
13. Lee, Y.C., Zomaya, A.Y.: Stretch out and compact: Workflow scheduling with resource abundance. In: Proceedings of 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 219–226. IEEE (2013)
14. Leslie, L.M., Sato, C., Lee, Y.C., Jiang, Q., Zomaya, A.Y.: DEWE: A framework for distributed elastic scientific workflow execution. In: Proceedings of 2015 13th Australasian Symposium on Parallel and Distributed Computing (AusPDC), pp. 3–10 (2015)
15. Maheshwari, K., Jung, E.S., Meng, J., Vishwanath, V., Kettimuthu, R.: Improving multisite workflow performance using model-based scheduling. In: Proceedings of 2014 43rd IEEE International Conference on Parallel Processing (ICPP), pp. 131–140. IEEE (2014)

16. Malawski, M.: Towards serverless execution of scientific workflows-hyperflow case study. In: Proceedings of 2016 11th Workshop on Workflows in Support of Large-Scale Science (WORKS@ SC), pp. 25–33 (2016)
17. Malawski, M., Juve, G., Deelman, E., Nabrzyski, J.: Algorithms for cost-and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds. Future Gener. Comput. Syst. **48**, 1–18 (2015)
18. Shams, K.S., Powell, M.W., et al.: Polyphony: a workflow orchestration framework for cloud computing. In: Proceedings of 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid), pp. 606–611 (2010)
19. Tanaka, M., Tatebe, O.: Disk cache-aware task scheduling for data-intensive and many-task workflow. In: Proceedings of 2014 IEEE International Conference on Cluster Computing (CLUSTER), pp. 167–175. IEEE (2014)