Serverless Execution of Scientific Workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions

Maciej Malawski, Adam Gajek, Adam Zima, Bartosz Balis, Kamil Figiela

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

Abstract

Scientific workflows consisting of a high number of interdependent tasks represent an important class of complex scientific applications. Recently, a new type of *serverless* infrastructures has emerged, represented by such services as Google Cloud Functions and AWS Lambda, also referred to as the Function-as-a-Service model. In this paper we take a look at such serverless infrastructures, which are designed mainly for processing background tasks of Web and Internet of Things applications, or event-driven stream processing. We evaluate their applicability to more compute- and data-intensive scientific workflows and discuss possible ways to repurpose serverless architectures for execution of scientific workflows. We have developed prototype workflow executor functions using AWS Lambda and Google Cloud Functions, coupled with the HyperFlow workflow engine. These functions can run workflow tasks in AWS and Google infrastructures, and feature such capabilities as data staging to/from S3 or Google Cloud Storage and execution of custom application binaries. We have successfully deployed and executed the Montage astronomy workflow, often used as a benchmark, and we report on initial results of its performance evaluation. Our findings indicate that the simple mode of operation makes this approach easy to use, although there are costs involved in preparing portable application binaries for execution in a remote environment.

While our solution is an early prototype, we find the presented approach highly promising. We also discuss possible future steps related to execution of scientific workflows in serverless infrastructures. Finally, we perform a cost analysis and discuss implications with regard to resource management for scientific applications in general.

Keywords: Scientific workflows, cloud functions, serverless architectures, FaaS

1. Introduction

Scientific workflows consisting of a large number of dependent tasks represent an important class of complex scientific applications that have been successfully deployed and executed in traditional cloud infrastructures, including Infrastructure as a Service (IaaS) clouds. Recently, a new type of *serverless* infrastructures has emerged, represented by such services as Google Cloud Functions (GCF) [1] or AWS Lambda [2]. This model is often called Function-as-a-Service (FaaS), an alternative to the well-known Infrastructure-as-a-Service (IaaS) model. These services allow deployment of software in the form of functions that are executed in the provider's infrastructure in response to specific events such as new files being uploaded to a cloud data store, messages arriving in queue systems or direct HTTP calls. This approach frees the user from having to maintain a server, including configuration and management of virtual machines, while resource management is provided by the platform in an automated and scalable way.

In this paper we take a look at such serverless infrastructures. Although designed mainly for processing background tasks of Web and Internet of Things applications, or event-driven stream processing, we nevertheless investigate whether they can be applied to more compute- and data-intensive scientific workflows. The main objectives of this paper are as follows:

- To present the main features of serverless infrastructures, comparing them to traditional infrastructure-as-a-service clouds,
- To discuss the options of using serverless infrastructures for execution of scientific workflows,
- To present our experience with a prototype implemented using the HyperFlow [3] workflow engine, AWS Lambda and Google Cloud Functions,
- To evaluate our approach using the Montage workflow [4], a real-world astronomy application,
- To discuss the costs and benefits of this approach, together with its implications for resource management of scientific workflows in emerging infrastructures.

This paper extends our earlier work presented at the WORKS workshop [5], where we reported the results achieved

Email address: `malawski@agh.edu.pl` (Maciej Malawski) using a prototype based on Google Cloud Functions only. Here we extend our prototype to support both GCF and AWS Lambda, and provide more detailed results of experiments performed on these platforms, more in-depth discussion of serverless platforms and workflow architectures, as well as a cost analysis comparing cloud functions to traditional IaaS clouds.

The paper is organized as follows. We begin with an overview of serverless infrastructures in Section 2. In Section 3 we propose and discuss alternative options for serverless architectures of scientific workflow systems. Our prototype implementation, based on HyperFlow, AWS Lambda and GCF, is described in Section 4. This is followed by evaluation using the Montage application, presented in Section 5. We discuss implications for resource management, including a sample cost analysis in Section 6, and present related work in Section 7. Section 8 provides a summary and description of future work.

2. Overview of serverless clouds

Writing "serverless" applications is a recent trend, mainly addressing Web and other event-driven distributed applications. It frees programmers from having to maintain a server – instead, they can use a set of existing cloud services directly from their application. Examples of such services include cloud databases such as Firebase or DynamoDB, messaging systems such as Google Cloud Pub/Sub, notification services such as Amazon SNS and so on. When there is a need to execute custom application code in the background, special "cloud functions" (hereafter simply referred to as functions) can be called. Examples of such functions are AWS Lambda, Google Cloud Functions (GCF) or Microsoft Azure Functions.

All these infrastructures are based on the functional programming paradigm: a function is a piece of software that can be deployed on the providers' cloud infrastructure and it performs a single operation in response to an external event.

Functions can be triggered by:

- an event generated by the cloud infrastructure, e.g. a change in a cloud database, a file being uploaded to a cloud object store, a new item appearing in a messaging system, or an action scheduled at a specified time,
- a direct request from the application via HTTP or cloud API calls.

The cloud infrastructure which hosts the functions is responsible for automatic provisioning of resources (including CPU, memory, network and temporary storage), automatic scaling when the number of function executions varies over time, as well as monitoring and logging. The user is responsible for providing executable code in a format required by the framework. Typically, the execution environment is limited to a set of supported languages: Node.js, Java and Python in the case of AWS Lambda, and Node.js in the case of GCF. The user has no control over the execution environment, such as underlying operating system, version of the runtime libraries, etc., but can use custom libraries with package managers and even upload binary code to be executed. A summary of features provisioned by main cloud function providers is shown in Table 1

Functions are thus different from Virtual Machines in IaaS clouds where the users have full control over the OS (including root access) and can customize the execution environment to their needs. On the other hand, functions free the developers from the need to configure, maintain, and manage server resources.

Cloud providers impose certain limits on the amount of resources a function can consume, as illustrated in Table 2. In the case of AWS Lambda these limits are as follows: temporary disk space: 512 MB, number of processes and threads: 1024, maximum execution duration per request: 300 seconds. There is also a limit of 100 concurrent executions per region, but this limit can be increased on request. GCF, in Beta since Feb. 2017, limits the concurrent executions to 400. There is also a timeout parameter that can be provided when deploying a function and the default value is 60 seconds. Azure functions have in general higher limits.

Functions are thus different from permanent and stateful services, since they are not long-running processes, but rather serve individual tasks. Resource limits indicate that such cloud functions are not currently suitable for large-scale HPC applications, but can be useful for high-throughput computing workflows consisting of many fine-grained tasks.

Functions have a fine-grained pricing model associated with them. In the case of AWS Lambda, the price is \$0.20 per 1 million requests and \$0.00001667 for every GB-second used, defined as CPU time multiplied by the amount of memory used. There are also additional charges for data transfer and storage (when DynamoDB or S3 is used). The beta version of Google Cloud Functions offers lower prices per execution time and free quota of requests per month, while Azure follows AWS pricing.

Functions can have a configurable size, e.g. their RAM allocation can be adjusted to 128, 256, 512, 1024 or 1536 MB in the case of AWS Lambda and Azure. GCF will also likely have a similar structure. Interestingly, AWS Lambda documentation states that the CPU, I/O and network allocation is proportional to RAM size. To the best of our knowledge, there are no good benchmark results available, however our initial results presented in Section 5 confirm that performance indeed depends on the function size.

Serverless infrastructures can be cost-effective compared to standard VMs. For example, the aggregate cost of running AWS Lambda functions with 1 GB of memory for 1 hour is \$0.060012. This is more expensive than the t2.micro instance, which also has 1 GB of RAM but costs \$0.013 per hour. A T2.micro instance, however, offers

Table 1: Main features of the leading cloud function providers

	AWS Lambda	***Google Cloud Functions***	***Azure Functions***
Language	Java, Python, Node.js	Node.js	Node.js, C#, F#
Pricing	\$0.20 per 1M requests and \$0.00001667/GB-s	\$0.40 per 1M requests (first two milions free) \$0.0000025/GB-s	\$0.20 per 1M requests and \$0.00001667/GB-s
Triggers	API Gateway, Event Sources (S3, SNS, SES, DynamoDB, Kinesis, CloudWatch)	Cloud Pub/Sub, Cloud Storage, Object Change Notifications	Schedule, HTTP, Azure Storage, Azure Event Hubs, Azure Service Bus
Deployment	Only zip upload	Zip + CVS (e.g. Git)	Zip + CVS (e.g. Git)
Versioning	Possible	No info	Not possible
Dependency Management	Not Possible	NPM	NPM and NuGet for C# and F#

Table 2: Limits of cloud function infrastructures

	AWS Lambda	***Google Cloud Functions***	***Azure Functions***
Execution Time	300s	540	No limit
Disk space	500 MB	Consumes memory resources	5 TB
Number of functions	No limit	1000	10
Parallel Execution	100 functions in parallel (configurable)	400	No limit

only burstable performance, which means only a fraction of CPU time per hour is available, e.g. T2.micro can use 100% of CPU capacity for only 10% of run time. The smallest standard instance at AWS is m3.medium, which costs \$0.067 per hour, but gives 3.75 GB of RAM. Both burstable and standard instances are billed at the beginning of each hour, while for cloud functions the billing interval is 100 ms, and hundreds of them can run in parallel, resulting in better elasticity. This means the provisioned capacity can quicker react to the current load, resulting in reduction of overprovisioning or underprovisioning. Cloud functions are thus more suitable for variable load conditions while standard instances can be more economical for applications with stable workloads.

In addition to the above-mentioned public cloud providers, there are initial Open Source solutions providing cloud functions that can be deployed on the premises. Iron.io created their own serverless technology called IronFunctions [6]. What distinguishes this solution from the ones discussed above is that it provides deployment using Docker containers, which can be hosted on Docker Hub or in our private Docker Trusted Registry. The flexibility of IronFunctions allowed Iron.io to adapt this solution to OpenStack, resulting in a framework called Picasso, which provides an API abstraction layer for serverless computing on OpenStack [7].

3. Execution of scientific workflows in serverless infrastructures

In light of the identified features and limitations of serverless infrastructures and cloud functions, we can discuss the option of using them for execution of scientific workflows. We will begin with a comparison of three workflow management system (WfMS) architectures (section 3.1), and then discuss several options for implementing a WfMS in a serverless infrastructure (section 3.2).

3.1. Overview of WfMS architectures

The first question is how the emergence of serverless infrastructures affects the architecture of workflow management systems. Fig. 1 presents simplified diagrams of three architectural types of scientific workflow systems: FaaS/serverless-based, service-based and “Resource” (IaaS)-based.

First, let us compare the FaaS-based architecture with the service-based one. The first similarity is that they are both inherently distributed – the call graph of the application is also the component graph. The second similarity is that some aspects of application management (e.g. server provisioning and application scaling) are in both cases done by an external entity (cloud provider or service

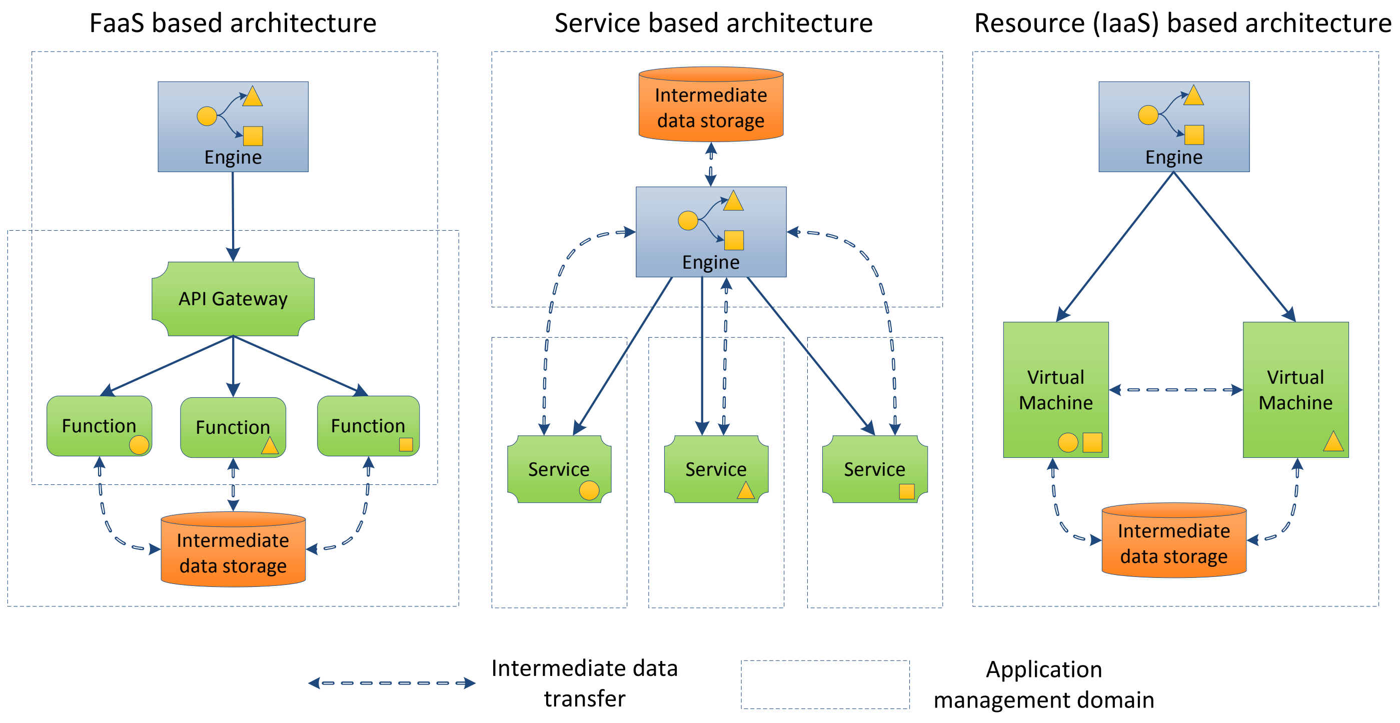
FaaS based architecture
Service based architecture
Resource (IaaS) based architecture
Engine
API Gateway
Function
Function
Function
Intermediate data storage
Intermediate data storage
Engine
Service
Service
Service
Engine
Virtual Machine
Virtual Machine
Intermediate data storage
Intermediate data transfer
Application management domain

Figure 1: Alternative architectures of scientific workflow management systems. Two questions that differentiate the architectures are emphasized: (1) Who is responsible for different aspects of application management? (2) How is intermediate data handled?

owner). These similarities are, however, arguably less consequential than the differences. First, unlike functions, services are deployed and maintained by different owners and normally are not under the control of the workflow developer. This leads to operational problems such as workflow decay [8], wherein a workflow can no longer be executed or produces different results than expected because the underlying services have changed. Workflow reproducibility is much easier with FaaS, because the workflow developer has full control over deployment of cloud functions. The second difference concerns management of intermediate data. Services are deployed in different infrastructures, so data transfer between them must be done via an external shared storage service which, in practice, is the memory of the workflow engine (possibly backed by a local database) [9]. This substantially affects the performance of data-intensive workflows. On the surface, FaaS-based architectures suffer from a similar problem because functions are stateless and their runtime environment is brought up and down for every request, so any data must also be passed through external persistent storage. However, in this case the functions can directly access the storage service, so the data does not have to be passed through the workflow engine. Moreover, the storage service is usually deployed in the same infrastructure as functions, so data transfers can be much more efficient, arguably making the FaaS architecture more suitable for data-intensive workflows. The third architectural type found in workflow systems is a "resource-based" one, wherein the workflow engine invokes application programs deployed on computing nodes of a distributed computing infrastructure such as a cluster or an IaaS cloud. Here, without sacrificing generality, we consider the latter case. In this architecture, the workflow owner is fully responsible for application management, including server provisioning, application deployment, mapping of application programs to computing nodes, scaling and fault tolerance. Intermediate data sometimes also must be transferred between nodes, but the options to achieve this are more diverse, including network file systems [10]. Moreover, only in this architecture can transfers of intermediate data be avoided altogether by mapping dependent tasks to the same nodes and storing intermediate data on local disks, or even in memory if the application components are designed to cache data between requests [11].

3.2. Options for workflow execution in serverless infrastructures

Having pointed out the specific characteristics of the FaaS-based WfMS architecture, let us discuss different options for its implementation in more detail. We will start with a traditional execution model in an IaaS cloud with no cloud functions (1), then present the queue model (2), direct executor model (3), bridge model (4), and decentralized model (5). These options are schematically depicted in Fig. 2, and discussed in detail further on.

3.2.1. Traditional model

The traditional model assumes the workflow runs in a standard IaaS cloud. In this model, workflow execution

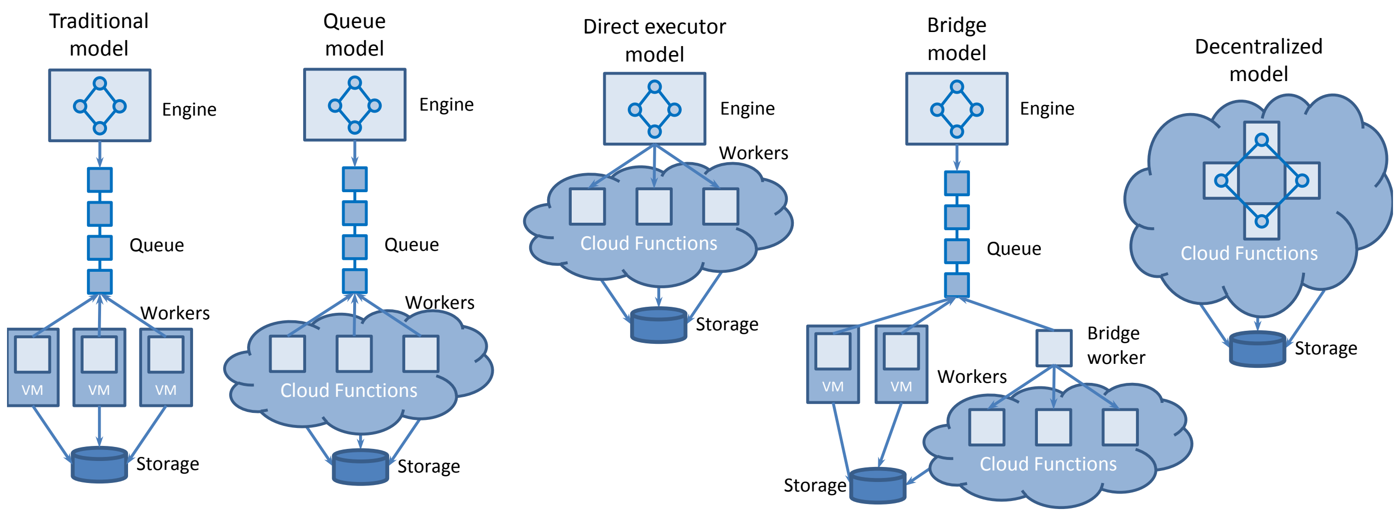

Figure 2: Options of serverless architectures for execution of scientific workflows.

follows the well-known master-worker architecture, where the master node runs a workflow engine, tasks that are ready for execution are submitted to a queue, and worker nodes process these tasks in parallel, whenever possible. The master node can be deployed in the cloud or outside of the cloud, while worker nodes are usually deployed as VMs in a cloud infrastructure. The worker pool is typically created on demand and can be dynamically scaled up or down depending on resource requirements.

Such a model is represented e.g. by Pegasus and HyperFlow. The Pegasus Workflow Management System [12] uses HTCondor [13] to maintain its queue and manage workers. HyperFlow [3] is a lightweight workflow engine based on Node.js – it uses RabbitMQ as its queue and AMQP Executors on worker nodes. The deployment options of HyperFlow on grids and clouds are discussed in detail in [14].

In this model the user is responsible for management of resources comprising the worker pool. The pool can be provisioned statically, which is commonly done in practice, but there is also ongoing research on automatic or dynamic resource provisioning for workflow applications [15, 16], which is a non-trivial task.

In the traditional cloud workflow processing model there is a need for some storage service to store input, output and temporary data. There are multiple options for data sharing [10], but one of the most widely used approaches is to rely on existing cloud storage, such as Amazon S3 or Google Cloud Storage. This option has the advantage of providing a permanent store so that data is not lost after the workflow execution is complete and the VMs are terminated.

3.2.2. Queue model

This model is similar to the traditional model: the master node and the queue remain unchanged, but the worker is replaced by a cloud function. Instead of running a pool of VMs with workers a set of cloud functions is prepared. Each task in a queue is translated to function call which returns the result via the queue.

The main advantage of this model is its simplicity, since it only requires changes in the worker module. This may be simple if the queue uses a standard protocol, such as AMQP in the case of HyperFlow Executor, but in the case of Pegasus and HTCondor a Condor daemon (condor_startd) must run on the worker node and communicate using a proprietary Condor protocol. In this scenario implementing a worker as a cloud function would require more effort.

Another advantage of the presented model is the ability to combine the workers implemented as functions with other workers running e.g. in a local cluster or in a traditional cloud. This would also enable concurrent usage of cloud functions from multiple providers (e.g. AWS and Google) when such a multi-cloud scenario is required.

An important issue associated with the queue model is how to trigger the execution of the functions. If a native implementation of the queue is used (e.g. RabbitMQ as in HyperFlow), it is necessary to trigger a function for each task added to the queue. This can be done by the workflow engine or by a dedicated queue monitoring service. Other options include periodic function execution or recursive execution: a function can itself trigger other functions once it finishes processing data.

To ensure a clean serverless architecture another option is to implement the queue using a native cloud service which is already integrated with cloud functions. In the case of AWS Lambda one could implement the queue using SQS or DynamoDB: here, a function could be triggered by adding a new item to a task table. In the case of GFC, a Google Cloud Pub/Sub service can be used for the same purpose. Such a solution, however, would require more changes in the workflow engine and would not be easy to deploy in multi-cloud scenarios.

3.2.3. Direct executor model

This is the simplest model and requires only a workflow engine and a cloud function that serves as a task executor. It eliminates the need for a queue since the workflow engine can trigger the cloud function directly via API/HTTP calls. Regarding development effort, it requires changes in the master and a new implementation of the executor: instead of a worker fetching the tasks from the queue the executor is implemented as a function.

Advantages of this model include its cleanness and simplicity, but these come at the cost of tight master-worker coupling. Accordingly, it becomes more difficult to implement the multi-cloud scenario, since the workflow engine would need to be able to dispatch tasks to multiple cloud function providers.

3.2.4. Bridge model

This solution is more complex but it preserves the decoupling of the master from the worker, using a queue. In this case the master and the queue remain unchanged, but a new type of bridge worker is added. It fetches tasks from the queue and dispatches them to the cloud functions. Such a worker needs to run as a separate service (daemon) and can trigger cloud functions using the provider-specific API.

The decoupling of the master from the worker allows for more complex and flexible scenarios, including multi-cloud deployments. A set of bridge workers can be spawned, each dispatching tasks to a different cloud function provider. Moreover, a pool of workers running in external distributed platforms, such as third-party clouds or clusters, can be used together with cloud functions.

3.2.5. Decentralized model

This model re-implements the whole workflow engine in a distributed way using cloud functions. Each task of a workflow is processed by a separate function. These functions can be triggered by (a) new data items uploaded to cloud storage, or (b) other cloud functions, i.e. predecessor tasks triggering their successor tasks following completion. Option (a) can be used to represent data dependencies in a workflow while option (b) can be used to represent control dependencies.

In the decentralized model the structure and state of workflow execution have to be preserved in the system. The system can be implemented in a fully distributed way, by deploying a unique function for each task in the workflow. In this way, the workflow structure is mapped to a set of functions and the execution state propagates by functions being triggered by their predecessors. Another option is to deploy a generic task executor function and maintain the workflow state in a database, possibly one provided as a cloud service.

The advantages of the decentralized approach include fully distributed and serverless execution, without the need to maintain a workflow engine. The required development effort is extensive, since it requires re-implementation of the whole workflow engine. A detailed design of such an engine is out of scope of this paper, but remains an interesting subject of future research.

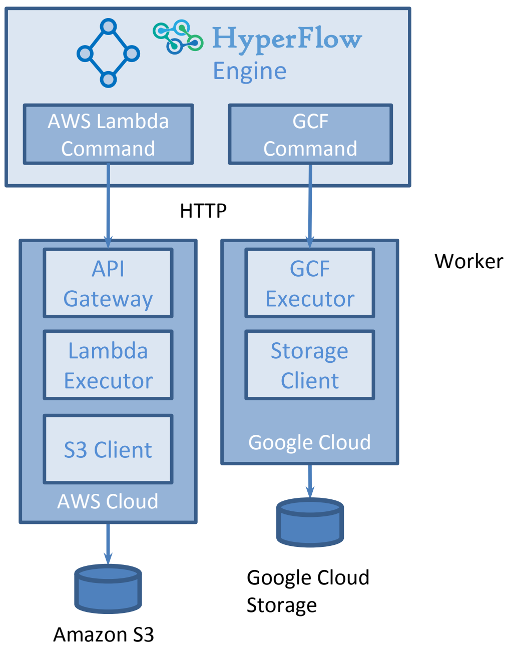

Figure 3: Architecture of the prototype integrating HyperFlow with AWS Lambda and Google Cloud Functions, using direct executor model

3.3. Summary of options

As we can see, cloud functions provide multiple integration options with scientific workflow engines. The users need to decide which option is best for them based on their requirements, most notably the allowed level of coupling between the workflow engine and the infrastructure and the need to run hybrid or cross-cloud deployments where resources from more than one provider are used in parallel. We consider the fully decentralized option as an interesting future research direction, while in the following sections we will focus on our experience with a prototype implemented using the direct executor model.

4. Prototype based on HyperFlow

To evaluate the feasibility of our approach we decided to develop a prototype using the HyperFlow engine and integrate it with two major cloud function providers: AWS Lambda and Google Cloud Functions, applying the direct executor model. This decision was made for several reasons. First, HyperFlow is implemented in Node.js, while cloud functions support Node.js as a native function execution environment. This good match simplifies development and debugging, which is always non-trivial in a distributed environment. Our selection of the direct execution model was motivated by the extensible design of

HyperFlow, which can associate with each task in a workflow a specific executor function responsible for handling command-line tasks. Since GCF provides a direct triggering mechanism for cloud functions using HTTP calls and AWS Lambda provides it via an HTTP API Gateway service, we can apply existing HTTP client libraries for Node.js, plugging support for cloud functions into HyperFlow as a natural extension. It should be noted that our prototype does not currently support using AWS Lambda and GCF in the same workflow execution, so each run has to specify which infrastructure to use.

4.1. Architecture and components

A schematic diagram of the prototype is shown in Fig. 3. We rely on the extension mechanism of HyperFlow, in which all the tasks (or processes) in the workflow are associated with a function responsible for task execution. These functions may e.g. call an external service, or submit the task to the execution queue, as in the traditional model which uses RabbitMQ, or execute tasks locally as the `command` function does. HyperFlow can be extended by providing new implementations of these functions.

On the engine side, we extended HyperFlow with two new functions: `AWS Lambda Command` and `GCF Command`, which are responsible for communication with cloud functions. They provide a replacement for the `AMQPCommand` function, which is used in the standard HyperFlow distributed deployment with the AMQP protocol and RabbitMQ. The role of command functions is to send the task description in a JSON-encoded message to the cloud function.

On the cloud function side, we implemented the `Lambda Executor` and `GCF Executor` functions, which need to be deployed on the AWS or GCF platforms respectively. The executor processes the message and uses the `Storage Client` for staging input and output data. It uses S3 or Google Cloud Storage respectively, and requests parallel transfers to speed up download and upload of data. The Executor calls the executable which needs to be deployed together with the function. Both AWS Lambda and GCF infrastructures support running custom Linux-based binaries, but the user has to make sure that the binary is portable, e.g. by statically linking all of its dependencies. Our architecture is thus purely serverless, with the HyperFlow engine running on a client machine and directly relying only on cloud services such as GCF and Cloud Storage.

In the case of AWS Lambda, we apply an architectural approach similar to GCF, but with a few additions. First, the `AWS Lambda Command` needs to communicate with AWS API Gateway to trigger our Lambda function. Moreover, since the API Gateway supports only API calls which take less than 30 seconds to complete, we needed to develop a simple retry mechanism to check if our Lambda function completed successfully. After receiving a timeout from API Gateway we start polling for proper output files on S3. In this way we can benefit from the AWS Lambda capability to run functions for up to 5 minutes.

4.2. Fault tolerance

Transient failures are a common risk in cloud environments. Since execution of a possibly large volume of concurrent HTTP requests in a distributed environment is always prone to errors caused by various layers of network and middleware stacks (load balancers, gateways, proxies, etc.), the execution engine needs to be able to handle such failures gracefully and attempt to retry failed requests.

In the case of HyperFlow, the Node.js ecosystem appears very helpful in this context. We used the `requestretry` library for implementing the HTTP client, which allows for automatic retry of failed requests with a configurable number of retries (default: 5) and delay between retries (default: 5 seconds). Our prototype uses these default settings, but in the future it will be possible to explore more advanced error handling policies taking into account error types and patterns.

5. Evaluation using Montage workflow

Based on our prototype which combines HyperFlow with AWS Lambda and Google Cloud Functions, we performed several experiments to evaluate our approach. The goals of the evaluation are as follows:

- To validate the feasibility of our approach, i.e. to determine whether it is practical to execute scientific workflows in serverless infrastructures.
- To measure performance characteristics of the execution environment in order to provide hints for resource management.

Details regarding our sample application, experiment setup and results are provided below.

5.1. Montage workflow and experiment setup

Montage application. For our study we selected the Montage [17] application, which is an astronomy workflow. It is often used for various benchmarks and performance evaluation, since it is open-source and has been widely studied by the research community. The application processes a set of input images from astronomic sky surveys and constructs a single large-scale mosaic image. The structure of the workflow is shown in Fig. 4: it consists of several stages which include parallel processing sections, reduction operations and sequential processing.

The size of the workflow, i.e. the number of tasks, depends on the size of the area of the target image, which is measured in angular degrees. For example, a small-scale 0.25-degree Montage workflow consists of 43 tasks, with 10 parallel mProjectPP tasks and 17 mDiffFit tasks, while more complex workflows can involve thousands of tasks. In our experiments we used the Montage 0.25 workflow with 43 tasks, the Montage 0.4 workflow with 107 tasks and Montage 0.6 with 165 tasks. Montage 0.6 was the largest workflow we were able to run on AWS Lambda, due to exceeding the 500 MB temporary disk space limit for larger workflows.

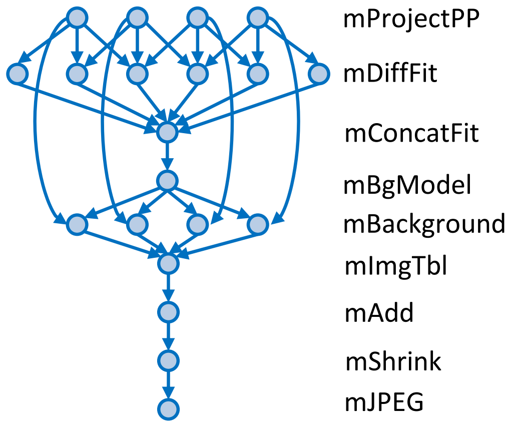

Figure 4: Structure of the Montage workflow used for experiments

Experiment setup. We used a recent version of HyperFlow, AWS Lambda and an Alpha version of Google Cloud Functions. The HyperFlow engine was installed on a client machine with Ubuntu 14.04 LTS Linux and Node.js 4.5.0. For staging the input and output data, as well as for temporary storage, we used an S3 and Google Cloud Storage bucket with standard options. For each cloud, the functions storage buckets were located in the same region `eu-central-1` for Lambda and `us-cental-1` for GCF, while the client machine was located in Europe.

Data preparation and handling. To run the Montage workflow in our experiments all input data needs to be uploaded to the cloud storage first. For each workflow run, a separate subfolder in the storage bucket is created. The subfolder is then used for exchange of intermediate data and for storing the final results. Data can be conveniently uploaded using a command-line tool which supports parallel transfers. The web-based AWS or Google console is useful for browsing results and displaying the resulting JPEG images.

5.2. Feasibility

To assess the feasibility of our approach we tested our prototype using the Montage 0.25 workflow, on both AWS Lambda and GCF. We collected task execution start and finish timestamps, which give the total duration of cloud function execution. This execution time also includes data transfers. Based on the collected execution traces we plotted Gantt charts. Altogether, several runs were performed and an example execution trace from GCF (representative of all runs) is shown in Fig. 5.

Montage 0.25 is a relatively small-scale workflow, but the resulting plot clearly reveals that the cloud function-based approach works well in this case. We can observe that the parallel tasks of the workflow (mProjectPP, mDiffFit and mBackground) are indeed quick and can be processed in parallel. The user has no control over the level of parallelism, but the cloud platform is able to process tasks in a scalable way, as stated in the documentation.

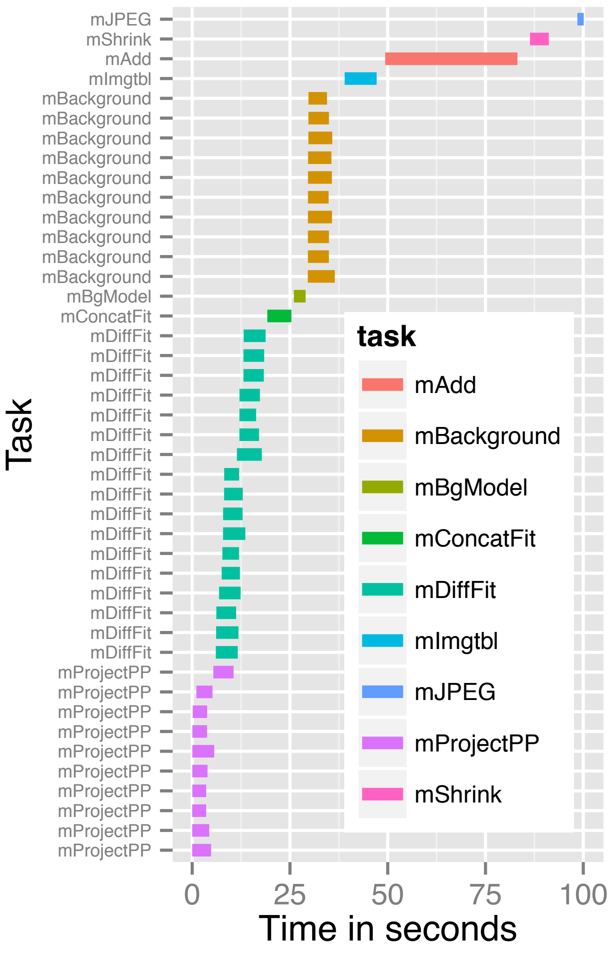

Figure 5: Sample run of Montage 0.25 workflow.

We also observe no significant delays between execution of tasks, and can attribute this to the fact that the requests between HyperFlow engine and the cloud functions are transmitted using HTTP over a wide-area network, including a trans-Atlantic connection.

Similar results were obtained for the Montage 0.4 workflow, which consists of 107 tasks; however the corresponding detailed plots are not reproduced here for reasons of readability. It should be noted that while the parallel tasks of Montage are relatively fine-grained, the execution time of sequential processing tasks such as mImgTbl and mAdd grows along with the size of the workflow and can exceed the default limit of 60 seconds imposed upon cloud function execution. This limit can be extended when deploying the cloud function, but there is currently no information regarding the maximum duration of such requests. We can only expect that such limits will increase as the platforms become more mature. This was indeed the case with Google App Engine, where the initial request limit was increased from 30 seconds to 10 minutes [18].

5.3. Deployment size and portability

Our current approach requires us to deploy cloud function together with all application binaries. AWS Lambda requires all code to be packaged during the deployment phase, while the Google Cloud Functions execution environment enables inclusion of dependencies in Node.js libraries packaged using the Node Package Manager (NPM) and automatically installed when the function is deployed. Moreover, in both infrastructures the user can provide a set of JavaScript source files, configuration and binary dependencies to be uploaded together with the function.

In the case of the Montage application, users need to prepare application binaries in a portable format. Since Montage is distributed in its source[1] format, it can be compiled and statically linked with all libraries making it portable to any Linux distribution.

The volume of the Montage binaries is 50 MB in total, and 20 MB in a compressed format, which is used for deployment. We consider this deployment size practical in most cases. If the size of executable becomes problematic, we may consider deploying one separate function per executable. We should note that deployment of the function is performed only once, prior to workflow execution. Of course, when the execution environment needs to instantiate the function or create multiple instances for scale-out scenarios, the size of each instance may affect performance, so users should try to minimize the volume of the deployment package. It is also worth noting that such binary distributions are usually more compact than full images of virtual machines used in traditional IaaS clouds. Unfortunately, if the source distribution or portable binary is not available, it may not be possible to deploy it as a cloud function. One useful option would be to allow deployment of container-based images, such as Docker images, but this is currently not supported.

5.4. Variability

Variability is an important metric of cloud infrastructures, since distribution and resource sharing often hamper consistent performance. To measure the variability of GCF while executing scientific workflows, we measured the duration of parallel task execution in the Montage (0.25 degree) workflow – specifically, mBackground, mDiffFit and mProjectPP – running 10 workflow instances over a period of one day.

Results are shown in Fig. 6. We can see that the distribution of tasks is moderately wide, with the inter-quartile range of about 1 second. The distribution is skewed towards longer execution times, up to 7 seconds, while the median is about 4 seconds. It is important that we do not observe any significant outliers. We have to note that the execution times of the tasks themselves vary (they are not identical) and that task duration includes data transfers to/from cloud storage. Having taken this into account,

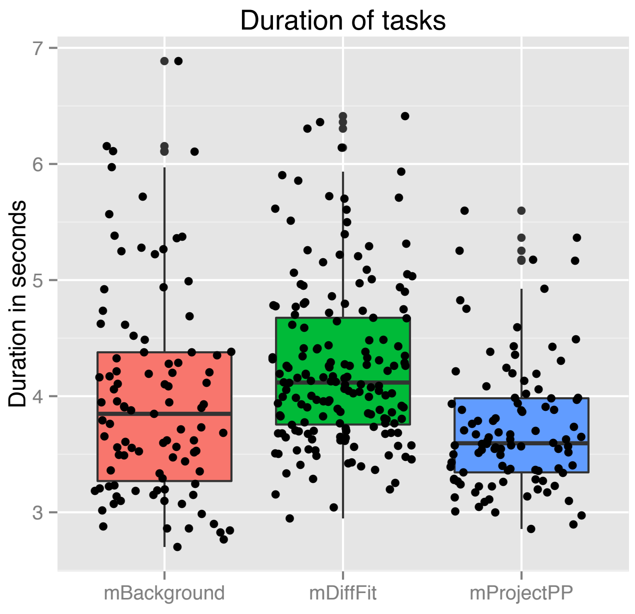

Figure 6: Distribution of execution times of parallel tasks of the Montage 0.25 workflow.

we can conclude that the execution environment behaves consistently in terms of performance, since the observed variation is rather low. Further studies and long-term monitoring would be required to determine whether such consistency is preserved over time.

5.5. Scalability

Cloud functions provide scalable and parallel execution, but the user has no control over the number of machines and over the assignment of tasks to machines. We can only try to infer the level of parallelism by analyzing the execution traces of workflows in which there are many independent tasks that can be executed in parallel. For these experiments we used Montage, in which all tasks of a given type can be executed at the same time in parallel. In our experiment, we ran the Montage 0.6 workflow on AWS Lambda and recorded task start/end timestamps. Then, based on these timestamps we assigned tasks to possible “machines” in such a way that there is no overlap, and to minimize the number of machines. In this way we obtained a potential assignment of tasks to a minimum set of machines required to execute a given workflow.

The results are shown in Fig. 7. We can see that the maximum parallelism achieved, as measured by the concurrency level, is over 60 (out of 100) on AWS Lambda. It is likely that more machines are actually needed to produce such an execution trace, since the delays between tasks are very small in this case. The reported number was consistent over several runs of this and other workflows, in some cases reaching 85 instances, which means that all mDiffFit tasks were running in parallel on distinct instances. We

[1]`http://montage.ipac.caltech.edu/`

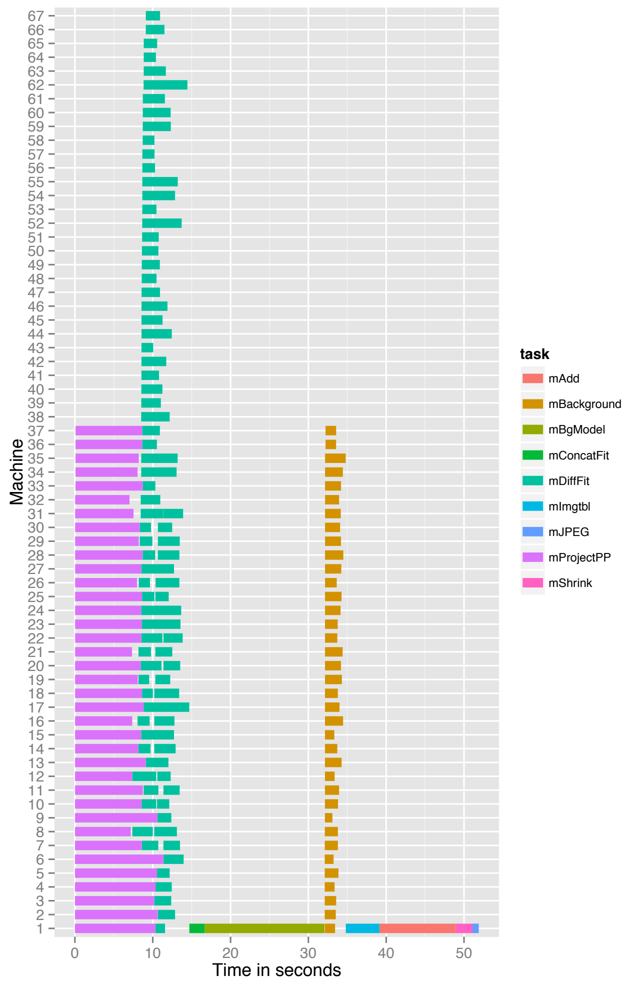

Figure 7: Possible distribution of tasks to machines based on a sample execution of the Montage 0.6 workflow on AWS Lambda with 1024 MB of RAM.

find these results very promising, revealing good scalability of AWS Lambda and potentially other FaaS platforms. At the time of writing we do not have the scalability results of the beta version of the GCF platform, and this will be the subject of future work as the platform goes into production.

5.6. Performance depending on function size

As mentioned in Section 2, AWS Lambda allows configuring the "size" of each function by setting its memory (RAM) allocation between 128MB and 1536MB in 64MB increments. According to the documentation, CPU and other resources are allocated proportionally to RAM size. To verify this experimentally, we ran the Montage 0.6 workflow with RAM set to 128, 256, 512, 1024 and 1536MB respectively.

Results are shown in Fig. 8, where we present the run times of parallel tasks: mBackground, mDiffFit and mProjectPP. Task duration comprises data transfer and execution time. We can observe that the execution time is not exactly inversely proportional to the function size, so performance does not scale linearly. Generally, the execution time decreases along with increases in RAM size, which is in agreement with the allocation policy described in the documentation. More detailed benchmarking will be necessary to precisely characterize the influence of CPU and I/O allocation on these results.

6. Discussion

Experiments conducted with our prototype implementation confirm the feasibility of our approach to execution of scientific workflows in serverless infrastructures. There are, however, some limitations that need to be emphasized here, and some interesting implications for resource management of scientific workflows in such infrastructures.

6.1. Granularity of tasks

Granularity of tasks is a crucial issue which determines whether a given application is well suited for processing using serverless infrastructures. It is obvious that for computationally heavy HPC applications a dedicated supercomputer is a better option. On the other hand, for high-throughput computing workloads distributed infrastructures such as grids and clouds have proven useful. Serverless infrastructures can be considered similar to these high-throughput infrastructures, but they usually have shorter task execution limits (300 seconds in the case of AWS Lambda and a 60-second default timeout for GCF). While these limits may vary, may be configurable or may change over time, we must assume that each infrastructure will always impose some kind of limit, which will constrain the types of supported workflows to those consisting of relatively fine-grained tasks. Many high-throughput workflows can fit into these constraints, but for the rest, other solutions should be developed, such as hybrid approaches.

6.2. Hybrid solutions

In addition to purely serverless solutions, we can propose hybrid approaches, such as the one outlined in Section 3. The presented bridge model is a typical hybrid solution which combines traditional VMs with cloud functions for lightweight tasks. This architecture can overcome the limitations of cloud functions, such as the need to create custom binaries or execution time limits.

The hybrid approach can also be used to minimize costs and optimize throughput. Such optimization should be based on cost analysis of leasing a VM and calling a cloud function, assuming that longer-term lease of resources typically corresponds to lower unit cost. This idea is generally applicable to hybrid cloud solutions [19]. For example, it may be more economical to lease VMs for long-running sequential parts of the workflow and trigger cloud functions for parallel stages, where spawning VMs that are billed on an hourly basis would be more costly. It may also prove interesting to combine cloud functions with spot instances

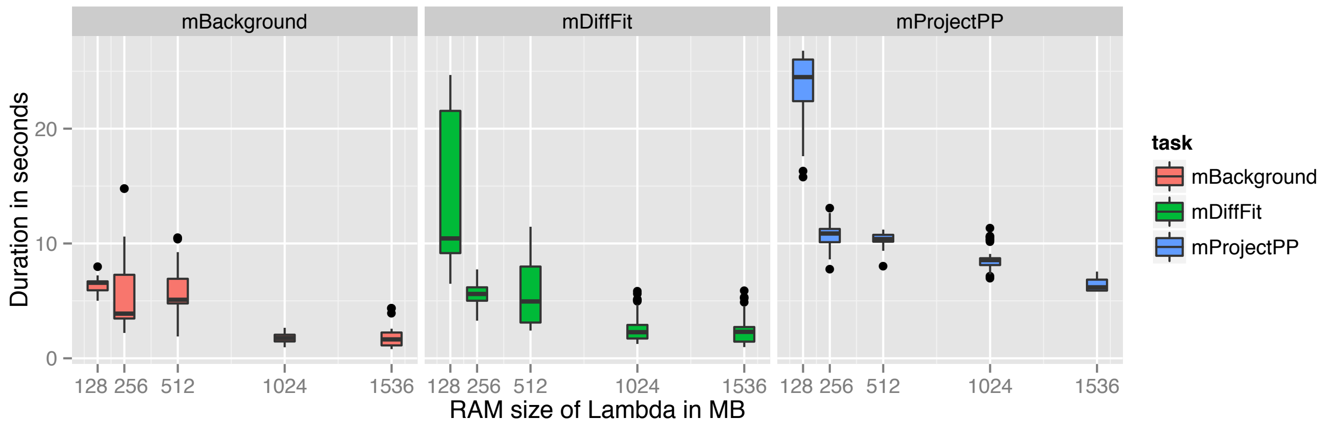

Figure 8: Distribution of execution times of parallel tasks of the Montage 0.6 workflow on AWS Lambda, depending on the configured RAM of the deployed function

or burstable [20] instances, which are cheaper but have varying performance and reliability characteristics.

The serverless approach may be used to speed up parallel parts of fork-join structured workflows. While sequential tasks, especially join tasks that deal with a large amount of data produced by the parallel part, would violate run time or disk space limits on serverless services and need to be processed in a classical environment, tasks in parallel stages are often fine-grained enough to fit within the function run time limit. While running such a workflow will still require a classical IaaS infrastructure, a significant amount of tasks from parallel stages can be offloaded to and massively parallelized on serverless platforms without the need for brief IaaS scale-out periods. Implementation-wise, the workflow tasks can be e.g. labeled with information regarding the target execution platform. Alternatively, a scheduler should dynamically decide, at runtime, whether the task being scheduled should be run as a function or using a classical worker, taking into account the task characteristics and infrastructure state (e.g. queue length for IaaS workers or data location).

The hybrid approach can also help resolve issues caused by the statelessness and transience of cloud functions, where no local data is preserved between function calls. By adding a traditional VM as one of the executor units, data transfers can be significantly reduced in the case of tasks that need to access to the same set of data multiple times.

6.3. *Resource management and autoscaling*

The core idea behind serverless infrastructures is that they free users from having to manage the server – and this also extends to clusters of servers. Decisions concerning resource management and autoscaling are thus made by the platform based on the current workload, history, etc. This is useful for typical Web or mobile applications that have interactive usage patterns and whose workload depends on user behavior. With regard to scientific workflows which have a well-defined structure, there is ongoing research on scheduling algorithms for clusters, grids and clouds. The goal of these algorithms it to optimize such criteria as time or cost of workflow execution, assuming that the user has some control over the infrastructure. In the case of serverless infrastructures the user does not have any control over the execution environment. The providers would need to change this policy by adding more control or the ability to specify user preferences regarding performance.

For example, users could specify priorities when deploying cloud functions, and a higher priority would mean a faster response time, quicker autoscaling, etc., but at an additional price. Lower-priority functions could have longer execution times, possibly relying on resource scavenging, but at a lower cost. Another option would be to allow users to provide hints regarding expected execution times or anticipated level of parallelism. Such information could be useful for internal resource managers to better optimize the execution environment and prepare for demand spikes, e.g. when many parallel tasks are launched by a workflow.

Adding support for cooperation between the application and the internal resource manager of the cloud platform would open an interesting area of research and optimization of applications and infrastructures which both users and providers could potentially benefit from.

6.4. *Cost analysis example*

The key question when considering serverless infrastructures is whether and when it may be more economical to use cloud functions or regular cloud instances. Here, we use Amazon services (EC2 and Lambda) to compare the cost of running a simplified scientific application. This analysis is a theoretical calculation based on the pricing model of AWS Lambda.

For this evaluation, we consider a bag-of-tasks application consisting of N independent tasks with a uniform task run time of 300 seconds. This can represent a parallel stage of a typical scientific workflow. 300 seconds (5 minutes) is the current limit of AWS Lambda. We impose a deadline constraint and calculate the cost of running this bag of tasks using (1) cloud functions (AWS Lambda) or (2) a traditional model with IaaS cloud instances. This corresponds to comparing the cost of the direct executor model to the traditional model presented in Fig. 2.

CPU allocation on Lambda is proportional to memory size of the function. Therefore, for CPU-bound tasks, the cost will remain constant regardless of memory size. This is caused by the fact that a function with a proportionally greater memory size will be proportionally more expensive, but, interestingly, will complete the task in proportionally less time, resulting in the same cost as a function with a lower memory capacity. Since the Lambda infrastructure is able to provide a high level of parallelism, we may assume that the cost of running the application remains constant. We assume that Lambda runs on the same hardware as EC2 instances, therefore task execution time remains fixed. We used real pricing for both services, valid as of January 2017, and relied on an `m4.large` instance (with 2 virtual cores) for comparison. We further assumed that tasks are single-threaded applications and the workflow execution engine uses all available cores of the VM. The cost of larger instances in the `m4` family is proportional to the number of virtual cores, so it does not influence results. For simplicity we also assumed infinite scalability of both Lambda and EC2 and no startup delays or other overheads.

Based on the number of tasks, their duration and the deadline, we calculated the number of IaaS instances required to run the application, and their cost using the hourly billing model. We also calculated the constant cost of running the application on the Lambda infrastructure. The consequence of this pricing model is that for some workloads cost of deployment to Lambda will be lower than to EC2.

The results of this calculation are presented in Fig. 9. We may observe that for sub-hour deadlines or small numbers of tasks Lambda is always cheaper than the classical EC2 infrastructure. This is a direct result of the hourly billing cycle in the case of IaaS. The serverless approach is definitely more suitable for workloads where a high level of parallelism is required for a short period of time. The cost of using IaaS instances is a step function of the deadline: steps are observed whenever an additional instance is needed. For example, with 20 tasks the total computing time is 100 minutes, or 50 minutes on 2-core machine. This means that for deadlines shorter than 50 minutes two instances are needed, while for longer deadlines a single instance suffices. However, in the case of larger workflows the traditional setup based on EC2 will be more cost efficient.

Nevertheless, even our simplified example shows that serverless infrastructures may be cost-efficient for compute-intensive applications in selected cases, depending on their characteristics.

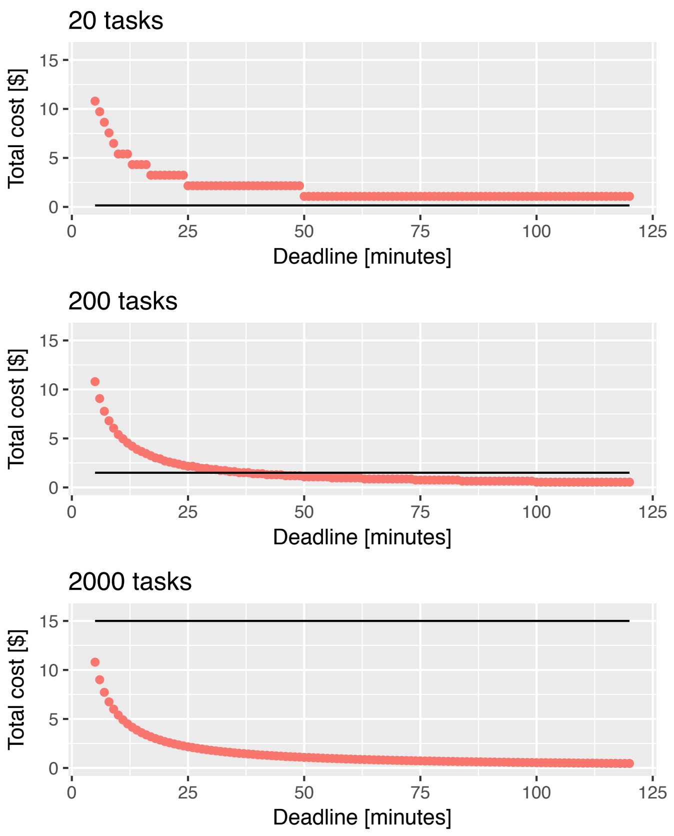

Figure 9: Cost of running the application on EC2 with a varying number of tasks and deadlines. The black line represents the cost of running the application on AWS Lambda, which is constant in our model.

7. Related Work

Although scientific workflows in clouds have been widely studied, research focus is typically on IaaS and little related work concerns serverless or other alternative types of infrastructures.

An example of using AWS Lambda for analyzing genomics data comes from the AWS blog [21]. The authors show how to use R, AWS Lambda and the AWS API gateway to process a large number of tasks. Their use case is to compute some statistics for every gene in the genome, which gives about 20,000 tasks in an embarrassingly parallel problem. This work is similar to ours, but our approach is more general, since we show how to implement generic support for scientific workflows.

A detailed performance and cost comparison of traditional clouds with microservices and the AWS Lambda serverless architecture is presented in [22]. An enterprise application was benchmarked and results show that serverless infrastructures can introduce significant savings without impacting performance. Similarly, in [4] the authors discuss the advantages of using cloud services and AWS Lambda for systems that require higher resilience. They show how serverless infrastructures can reduce costs in comparison to traditional IaaS resources and the spot market. An interesting discussion of serverless and cloud event programming paradigms is given in [23], where the case studies are blogging and media management application. Although these use cases are different from our scientific scenario, we believe that serverless infrastructures offer an interesting option for scientific workflows.

An interesting general discussion on the economics of hybrid clouds is presented in [19]. The author shows that even if when a private cloud is strictly cheaper (per unit) than public clouds, a hybrid solution can result in a lower overall cost in the case of variable workload. We expect that a similar effect can be observed in the case of a hybrid solution combining traditional and serverless infrastructures for scientific applications which often have a wide range of granularity of tasks.

Regarding the use of alternative cloud solutions for scientific applications, there is work on evaluation of Google App Engine for scientific applications [24, 18]. Google App Engine is a Platform-as-a-Service cloud, designed mostly for Web applications, but with additional support for processing background tasks. App Engine can be used for running parameter-study high-throughput computing workloads, and there are similar task processing time limits as in the case of serverless infrastructures. The difference is that the execution environment is more constrained, e.g. only one application framework is allowed (such as Java or Python) and there is no support for native code and access to local disk. For these reasons, we consider cloud functions such as AWS Lambda or Google Cloud Functions as a more interesting option for scientific applications.

The concept of cloud functions can be considered an evolution of earlier remote procedure call concepts, such as GridRPC [25], proposed and standardized for Grid computing. The difference between these solutions and current cloud functions is that the latter are supported by commercial cloud providers with emphasis on ease of use and development productivity. Moreover, the granularity of tasks processed by current cloud functions tends to be finer, so we need to follow the development of these technologies to further assess their applicability to scientific workflows.

From another perspective, cloud functions are a natural evolution of early approaches to executing programs over the Web interface, as defined in the widely used Common gateway Interface (CGI) standard [26]. CGI enables a Web server to run command-line programs in order to return dynamic Web pages. As such, our usage of cloud functions to execute program binaries is very similar, but of course the difference is in the way the infrastructure manages scalability and resource provisioning. Current cloud infrastructures are increasingly elastic, which was not the case with early Web servers running CGI programs.

A recently developed approach to decentralized workflow execution in clouds is represented by Flowbster [27], which also focuses on serverless infrastructures. We can expect that more similar solutions will emerge in the near future.

The architectural concepts of scientific workflows are discussed in the context of component and service architectures [28]. Cloud functions can be considered a specific class of services or components, which are stateless and can be deployed in cloud infrastructures. They do not impose any rules of composition, giving more freedom to developers. The most important distinction is that they are backed by the cloud infrastructure which is responsible for automatic resource provisioning and scaling.

The architectures of cloud workflow systems are also discussed in [29]. We believe that such architectures need to be re-examined as new serverless infrastructures become more widespread.

Based on the discussion of related work we conclude that our paper is likely the first attempt to use serverless clouds for scientific workflows and we expect that more research in this area will be needed as platforms become more mature.

8. Summary and future work

In this paper we presented our approach to combining scientific workflows with the emerging serverless clouds. We believe that such infrastructures based on the concept of cloud functions, such as AWS Lambda or Google Cloud Functions, provide an interesting alternative not only for typical enterprise applications, but also for scientific workflows. We have discussed several options for designing serverless workflow execution architectures, including queue-based, direct executor, hybrid (bridged) and decentralized ones.

To evaluate the feasibility of our approach we implemented a prototype based on the HyperFlow engine coupled with AWS Lambda and Google Cloud Functions. The prototype was evaluated with the real-world Montage application. Experiments with small-scale workflows consisting of between 43 and 165 tasks confirm that both AWS Lambda and GCF platforms can be successfully used, and that this process does not introduce significant delays. We have to note that the application needs to made portable in order to facilitate execution on such infrastructures, and that this may present an issue for more complex scientific software packages. Our preliminary results confirm good scalability of the AWS Lambda infrastructure within the limit of 100 concurrent invocations. We also measured how the performance of AWS Lambda functions depends on the allocated memory for our application.

Our paper also presents some implications of serverless infrastructures for resource management of scientific workflows. First, we observe that not all workloads are suitable due to execution time limits, e.g. 5 minutes in the case of AWS Lambda – accordingly, the granularity of tasks has to be taken into account. We discuss how hybrid solutions combining serverless and traditional infrastructures can help optimize the performance and cost of scientific workflows. We also suggest that adding more control or the ability to provide priorities or hints to cloud platforms could benefit both providers and users in terms of optimizing performance and cost. The cost analysis we preformed suggests that using cloud functions may be more cost-effective than relying on traditional IaaS clouds, depending on application characteristics such as the number of tasks and execution deadlines.

Since this is a fairly new topic, we see many options for future work. Further implementation and evaluation of various serverless architectures for scientific workflows is needed, with the decentralized option regarded as the greatest challenge. A more detailed performance evaluation of different classes of applications on various emerging infrastructures would also prove useful to better understand the possibilities and limitations of this approach. Finally, interesting research can be conducted in the field of resource management for scientific workflows, in order to propose strategies and algorithms for optimizing the time or cost of workflow execution in the emerging serverless clouds.

Acknowledgments

This work is partially supported by the National Science Centre, Poland, grant 2016/21/B/ST6/01497. Maciej Malawski would like to thank the Google Cloud Functions team for the opportunity to use the alpha version of their service.

References

[1] Cloud Functions - Serverless Microservices — Google Cloud Platform, `https://cloud.google.com/functions/` (2016).

[2] AWS Lambda - Serverless Compute, `https://aws.amazon.com/lambda/` (2016).

[3] B. Balis, HyperFlow: A model of computation, programming approach and enactment engine for complex distributed workflows, Future Generation Computer Systems 55 (2016) 147–162. doi:10.1016/j.future.2015.08.015.
URL http://www.sciencedirect.com/science/article/pii/S0167739X15002770

[4] B. Wagner, A. Sood, Economics of Resilient Cloud Services, in: 1st IEEE International Workshop on Cyber Resilience Economics, 2016. arXiv:1607.08508.
URL http://arxiv.org/abs/1607.08508

[5] M. Malawski, Towards serverless execution of scientific workflows – HyperFlow case study, in: WORKS 2016 Workshop, Workflows in Support of Large-Scale Science, in conjunction with SC16 Conference, CEUR-WS.org, Salt Lake City, Utah, USA, 2016.

[6] IronFunctions — Iron.io, `https://github.com/iron-io/functions` (2017).

[7] OpenStack Picasso Functions as a Service, `https://github.com/openstack/picasso` (2017).

[8] J. Zhao, J. M. Gomez-Perez, K. Belhajjame, G. Klyne, E. Garcia-Cuesta, A. Garrido, K. Hettne, M. Roos, D. De Roure, C. Goble, Why workflows breakunderstanding and combating decay in taverna workflows, in: E-Science (e-Science), 2012 IEEE 8th International Conference on, IEEE, 2012, pp. 1–9.

[9] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, C. Goble, Taverna, reloaded, in: International conference on scientific and statistical database management, Springer, 2010, pp. 471–481.

[10] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, P. Maechling, Data Sharing Options for Scientific Workflows on Amazon EC2, in: SC '10 Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, IEEE Computer Society, 2010, pp. 1–9. doi:10.1109/SC.2010.17.
URL http://portal.acm.org/citation.cfm?id=1884693

[11] B. Balis, K. Figiela, K. Jopek, M. Malawski, M. Pawlik, Porting hpc applications to the cloud: A multi-frontal solver case study, Journal of Computational Science 18 (2017) 106–116.

[12] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, K. Wenger, Pegasus, a workflow management system for science automation, Future Generation Computer Systems 46 (2015) 17–35. doi:10.1016/j.future.2014.10.008.
URL http://www.sciencedirect.com/science/article/pii/S0167739X14002015

[13] D. Thain, T. Tannenbaum, M. Livny, Distributed computing in practice: the Condor experience, Concurrency and Computation: Practice and Experience 17 (2-4) (2005) 323–356. doi:doi:10.1002/cpe.938.
URL citeulike-article-id:866823http://dx.doi.org/10.1002/cpe.938

[14] B. Balis, K. Figiela, M. Malawski, M. Pawlik, M. Bubak, A Lightweight Approach for Deployment of Scientific Workflows in Cloud Infrastructures, in: R. Wyrzykowski, E. Deelman, J. Dongarra, K. Karczewski, J. Kitowski, K. Wiatr (Eds.), Parallel Processing and Applied Mathematics: 11th International Conference, PPAM 2015, Krakow, Poland, September 6-9, 2015. Revised Selected Papers, Part I, Springer International Publishing, Cham, 2016, pp. 281–290. doi:10.1007/978-3-319-32149-3_27.
URL http://dx.doi.org/10.1007/978-3-319-32149-3{_}27

[15] M. Malawski, G. Juve, E. Deelman, J. Nabrzyski, Algorithms for cost-and deadline-constrained provisioning for scientific workflow ensembles in IaaS clouds, Future Generation Computer Systems 48 (2015) 1–18. doi:10.1016/j.future.2015.01.004.

[16] M. Mao, M. Humphrey, Auto-scaling to minimize cost and meet application deadlines in cloud workflows, in: SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, ACM, Seattle, Washington, 2011. doi:10.1145/2063384.2063449.
URL http://portal.acm.org/citation.cfm?id=2063384.2063449

[17] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. Prince, Others, Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking, International Journal of Computational Science and Engineering 4 (2) (2009) 73–87.

[18] M. Malawski, M. Kuzniar, P. Wojcik, M. Bubak, How to Use Google App Engine for Free Computing, IEEE Internet Computing 17 (1) (2013) 50–59. doi:10.1109/MIC.2011.143.
URL http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp={&}arnumber=6065729

[19] J. Weinman, Hybrid Cloud Economics, IEEE Cloud Computing 3 (1) (2016) 18–22. doi:10.1109/MCC.2016.27.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7420473

[20] P. Leitner, J. Scheuner, Bursting with Possibilities – An Empirical Study of Credit-Based Bursting Cloud Instance Types (dec 2015). doi:10.1109/UCC.2015.39.
URL https://www.computer.org/csdl/proceedings/ucc/2015/5697/00/5697a227-abs.html

[21] Bryan Liston, Analyzing Genomics Data at Scale using R, AWS Lambda, and Amazon API Gateway — AWS Compute Blog, `http://tinyurl.com/h7vyboo` (2016).

[22] M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, M. Lang, Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and

microservice architectures, in: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2016, pp. 179–182. doi:10.1109/CCGrid.2016.37.

[23] G. McGrath, B. Judson, P. Brenner, J. Short, S. Ennis, Cloud event programming paradigms: Applications and analysis, in: EEE Cloud 2016 Conference, 2016.

[24] R. Prodan, M. Sperk, S. Ostermann, Evaluating High-Performance Computing on Google App Engine, IEEE Software 29 (2) (2012) 52–58. doi:10.1109/MS.2011.131.
URL http://www.computer.org/csdl/mags/so/2012/02/mso2012020052.html

[25] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, H. Casanova, Overview of gridrpc: A remote procedure call api for grid computing, in: International Workshop on Grid Computing, Springer, 2002, pp. 274–278.

[26] D. Robinson, K. A. L. Coar, The Common Gateway Interface (CGI) Version 1.1.
URL https://tools.ietf.org/html/rfc3875

[27] P. Kacsuk, J. Kovacs, Z. Farkas, Flowbster: Dynamic creation of data pipelines in clouds, in: Digital Infrastructures for Research event, Krakow, Poland, 28-30 September 2016, 2016.

[28] D. Gannon, Component Architectures and Services: From Application Construction to Scientific Workflows, Springer London, London, 2007, pp. 174–189. doi:10.1007/978-1-84628-757-2_12.
URL http://dx.doi.org/10.1007/978-1-84628-757-2_12

[29] X. Liu, D. Yuan, G. Zhang, W. Li, D. Cao, Q. He, J. Chen, Y. Yang, The Design of Cloud Workflow Systems, Springer New York, New York, NY, 2012. doi:10.1007/978-1-4614-1933-4.