Handling heterogeneous workflows in the Cloud while enhancing optimizations and performance

Emile Cadorel, Hélène Coullon, Jean-Marc Menaud

Handling heterogeneous workflows in the Cloud while enhancing optimizations and performance

Emile Cadorel
Spirals, INRIA, Univ. Lille, CRIStAL
Lille, France
emile.cadorel@inria.fr

Hélène Coullon
IMT Atlantique, INRIA, LS2N
Nantes, France
helene.coullon@imt-atlantique.fr

Jean-Marc Menaud
IMT Atlantique, INRIA, LS2N
Nantes, France
jean-marc.menaud@imt-atlantique.fr

***Abstract*—The goal of a workflow engine is to facilitate the writing, the deploying, and the execution of a scientific workflow (*i.e.*, graph of coarse-grain and heterogeneous tasks) on distributed infrastructures. With the democratization of the Cloud paradigm, many workflow engines of the state of the art offer a way to execute workflows on distant data centers by using the Infrastructure-as-a-Service (IaaS) or the Function-as-a-Service (FaaS) services of Cloud providers. Hence, workflow engines can take advantage of the (presumably) infinite resources and the economical model of the Cloud. However, two important limitations lie in this vision of Cloud-oriented workflow engines. First, by using existing services of Cloud providers, and by managing the workflows at the user side, the Cloud providers are unaware of both the workflows and their user needs, and cannot apply specific resource optimizations to their infrastructure. Second, for the same reasons, handling the heterogeneity of tasks (different operating systems) in workflows necessarily degrades either the transparency for the users (who must provision different types of resources), or the completion time performance of the workflows, because of the stacking of virtualization layers. In this paper, we tackle these two limitations by presenting a new Cloud service dedicated to scientific workflows. Unlike existing workflow engines, this service is deployed and managed by the Cloud providers, and enables specific resource optimizations and offers a better control of the heterogeneity of the workflows. We evaluate our new service in comparison to Argo, a well-known workflow engine of the literature based on FaaS services. This evaluation was made on a real distributed experimental platform with a realistic and complex scenario.**

I. Introduction

Scientific workflows model scientific applications as a set of coarse-grain tasks linked together through data dependencies. Developing scientific applications as a set of interdependent tasks is a common pattern that has been adopted by many scientists. This approach allows to develop complex applications by splitting them into different simpler parts, which enhances their development and maintainability as well as their parallelization and distribution. However, such workflows often become very complex and difficult to manage, deploy and execute for scientists. To simplify the management of workflows the scientists generally use workflow engines, such as Pegasus [1], DEWE [2], Airflow [3], Argo [4], or Hyperflow [5]. In this paper we aim at tackling two drawbacks of existing engines.

First, these engines are generally designed to be used and deployed by the scientists (*i.e.*, end-users) on existing Cloud services, and are then used to automatically orchestrate the execution of the tasks of a given workflow for a given user. For instance, Pegasus and Hyperflow use Infrastructure-as-a-Service (IaaS) and DEWE uses Function-as-a-Service (FaaS). This conceptual design imposes that the workflow engine runs on the end-users side, thus that the Cloud provider is fully unaware of the workflows. For this reason, Cloud providers cannot take advantage of the workflows to optimally schedule tasks of users on their resources. Hence, scheduling algorithms of the literature [6]–[11] cannot be leveraged by Cloud providers to optimize, for instance, their costs and energy consumption, or to favor the fairness between users etc.

Second, the management of a potential extreme heterogeneity of tasks in workflows is limited in existing engines. Indeed, in addition to the different library requirements, some tasks may need to be executed on different operating systems (OS) in the same workflow. For example, the genomic data stream designed by the ICO in [12]–[14] uses data produced by a vendor-specific machine[1]. To convert the output formats of the machine to standards, a specific software developed for Windows must be used, while the other libraries (*e.g.*, Openswath) must be run under Linux. This is a common issue in scientific workflows. None of the existing workflow engines offer this level of heterogeneity while not degrading one of the following aspects: the performance of the workflow execution (*i.e.*, completion time, power efficiency); or the simplicity of usage for the scientist.

In this document, we present the definition and implementation of a service, specifically designed for the execution of scientific workflows. By moving the workflow management from the scientist side to the Cloud provider side, and thanks to its modular and distributed design, our new service solves the two following aspects compared to the state of the art: (i) handling heterogeneous workflows without loss of performance nor simplicity for the end-user (*i.e.*, the scientist); (ii) enabling workflow-specific resources optimizations for Cloud providers (*e.g.*, cost and energy savings).

The remainder of this paper is organized as follows. First, Section II presents the related work on workflow execution and workflow engines, and highlights the limitations of existing solutions. Second, Section III details the contribution: (i) the

[1]Data acquisition documentation

	Scientist concerns	*References*
1	Easy workflow specification	[1], [3]–[5], [15]–[17]
2	Automatic execution	[1], [3]–[5], [15]–[17]
3	Transparent provisioning	[3], [4], [16], [17]
4	Transparent elasticity	[3], [4], [15]–[17]

TABLE I: Concerns of the scientists (*i.e.*, end-users) when using a workflow engine.

workflow submission language for the end-users, and (ii) the architecture and the modular aspect of the new service for the Cloud provider. Third, Section IV evaluates this solution. Finally, Section V concludes this works and opens some perspectives.

II. Background and Related Work

From a scientist (*i.e.*, end-user) perspective, two important features have to be considered for a workflow engine: being able to easily specify a scientific workflow; and being able to submit a workflow which is automatically executed while handling software dependencies and files (*i.e.*, data) between tasks. This is of course the basic purpose of any workflow engine found in the literature [1], [3]–[5], [15]–[17]. In addition to this, in the context of Cloud computing, provisioning (*i.e.*, booking) resources on Cloud infrastructures should be fully transparent for the end-user who is not a DevOps engineer, nor a developer in most cases. Finally, the end-user may want the elasticity of the provisioned resources to be handled automatically, to reduce their costs when executing a workflow. Indeed, because workflows are composed of many inter-dependent tasks with heterogeneous execution time and requirements, the number of required resources changes during their execution. By dynamically adapting the provisioned resources on the Cloud (*i.e.*, elasticity), the overall cost is reduced which is important for the scientist who may run hundreds of workflows. Of course, the elasticity should be handled automatically and transparently from the end-user viewpoint. These four requirements for scientists are summarized in Table I.

Pegasus [1] and Hyperflow [5] are two well-known engines that are deployable on resources reserved by an end-user. To use Pegasus or Hyperflow on Cloud infrastructures, an end-user has to reserve a cluster of virtual machines (VMs) in an IaaS Cloud service (Infrastructure-as-a-Service), install the workflow engine on these virtual resources, and launch the execution of the workflow with the engine. On the one hand, they require from the end-users the ability to deploy an infrastructure by themselves, assuming that the end-users composing the workflows, *i.e.*, scientists, are experts in resource management. Moreover, they also expect from the end-users to manage the various dependencies required by the workflow engine to be properly configured and executed. In addition to being difficult and time-consuming, this limits the possibility of re-using the efforts between several users. Indeed, in addition to the workflow specification, some deployment and configuration scripts should be sharable between scientists but may have to be adapted to different infrastructures and user requirements. If DevOps tools could help in this task (*e.g.*, Terraform, Ansible, Puppet etc.) this remains an important time-consuming and technical drawback for scientists. This may require that an expert IT engineer work with the scientists. On the other hand, as the end-users are responsible for resource provisioning on the Cloud, they have to determine the number of resources that will be needed for the overall execution of their workflows. This is a very difficult task that can lead to under-used resources. Furthermore, the resource usage of a workflow may vary during its execution lifetime (*e.g.*, the number of parallel tasks at each step is variable), and therefore a non elastic resource reservation seems inappropriate.

	Heterogeneity	*Scientist (Table I)*	*Performance*	*Optimizations (providers)*
Airflow [3]	✓	✗	✓	✗
	✗	✓	✓	✗
Argo [4]	✓	✓	✗	✗
	✗	✓	✓	✗

TABLE II: Metrics of interest on Airflow and Argo. None of these solutions succeed in handling heterogeneous workflows while preserving both the scientists concerns and a good performance when executing workflows. None of these solutions enable specific resources optimizations from a Cloud provider viewpoint.

Some workflow engines leverage the usage of the FaaS (Function as a Service) Cloud services [3], [4], [15]–[18]. The FaaS offers the opportunity to define functions (small computing entities using specific libraries), that will be executed in a serverless environment. The idea of workflow engines coupled to a FaaS service is to associate one task of a workflow to one function in the FaaS. In the FaaS, the deployment and management of resources are handled by the Cloud provider, and are hidden from the end-user. Basically, the resources provisioned in a FaaS are containers. By using the FaaS, the resources are provisioned when required, and released when no longer used, therefore enhancing the transparent elasticity of the solution. To the best of our knowledge, as indicated in Table I, only four FaaS-oriented workflow engines of the literature offer the four expected properties from the scientist perspective [3], [4], [16], [17]. Because Apache Airflow, and Argo have accessible documentations and implementations, we study more specifically these two solutions in the rest of this section.

An important aspect of the contribution presented in this article is to design a solution capable of managing and executing highly heterogeneous workflows containing tasks that need to be run on different operating systems. This property is difficult to address when combined with two other properties : (1) preserving the scientist properties of Table I; and (2) avoiding a loss of performance in the completion time of workflows. Table II resumes the combination of these three properties, or metrics, for both Airflow and Argo. In these workflow engines, handling heterogeneous workflows has to be done with some compromises. First, in Airflow it is possible to define specific provisioners for any type of resource, from bare-metal resources to virtual machines. However, this functionality degrades the provisioning transparency

for the scientist. Indeed, to use heterogeneity, the end-user has to provide supplementary technical script files. Second, because it is relying on a Kubernetes cluster, Argo is based on containers and in particular Docker containers. If multiple operating systems are required by the tasks, a virtual machine (or a specific additional subsystems as WSL) and a container have to be stacked which involves a high overhead, reducing the performance of the execution. Furthermore, bare-metal resources are not available in Argo. The solution presented in [16], [17] have the same limitations as Airflow, as they lie on public FaaS services (*e.g.*, AWS Lambda, IBM Cloud functions), and does not manage transparently the heterogeneity.

Finally, and this is the main point of this paper, in both Airflow and Argo (but also in other engines of the literature) the Cloud provider cannot make specific optimizations to handle scientific workflows. Indeed, generic Cloud services (*i.e.*, IaaS and FaaS) are directly used by workflow engines, thus leaving the Cloud provider outside the decisions of resource allocation and scheduling. This is potentially an economical loss for the Cloud provider who would have been able to perform smart global optimizations by considering all submitted workflows of all users.

As a matter of fact, existing engines are designed to run one workflow of one end-user, and therefore make local optimizations like for example makespan minimization [1], [5], or cost optimization [3], [4], [19], [20], without any knowledge of the physical infrastructure, while assuming that an infinite number of resources are available. Yet, when many users share the same infrastructure, applying a set of local optimizations, at the level of one workflow, leads to a sub-optimal utilization of the infrastructure in many dimensions (*e.g.*, fairness, energy efficiency, etc.). In particular, by being unaware of the workflows, the Cloud provider cannot leverage the optimization algorithms of the literature [6]–[11].

In this paper we aim at tackling the execution of heterogeneous workflows in the Cloud while preserving the scientist concerns and execution performance. Furthermore, by offering a workflow-specific service deployed at the Cloud provider side and designed to be modular with an easy way to plug new types of resources or schedulers, we also enable optimizations at the resource management level. As preserving the concerns of the scientist is a primary issue in our proposal, a comparison of our new service with Argo is made in the evaluation in Section IV.

III. WaaS : Workflow as a Service

In this section is presented our new Cloud service dedicated to scientific workflows, namely the WaaS for Workflow-as-a-Service. The idea of the WaaS is, unlike usual workflow engines, to provide a solution for scientific workflows deployed and managed on the Cloud provider side. This way, the limitations of the state of the art that we pointed out in the previous section can be solved: heterogeneity and optimizations of the resources.

This section is divided into two parts. The first part presents the operations performed by the end-user (*i.e.*, the scientist) wishing to run a scientific workflow in the WaaS. The second part presents how the service can be deployed and customized by a Cloud provider. This second part also presents a detailed view of the service architecture and its different modules.

```
files: # list of files
    - id: value # file ID
      name: value # file name
      type: output # Optional
      *additional optional attributes
tasks: # list of tasks
    - app: value # Name of executable
      hardware: # list of Hardware requirements
        key1: value1
        key2: value2
        ...
      os:
        name : value # Name of operating system
        software: # list of software dependencies
          - value1
          - ...
      output: # list of file references
        - value1
        - ...
      input: # list of file references
        - value1
        - ...
      params : value # execution parameters
      *additional optional attributes
```

Fig. 1: Meta-grammar of the workflow description file

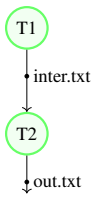
A. Scientist concerns

As presented in the previous section, a scientific workflow is a succession of tasks with file dependencies, which can be described by a DAG (directed acyclic graph). In our context, the job of the end-user is to select the different tasks of the workflow and to describe the workflow topology by composing them. The tasks composing the workflows are usually developed by experts in the community of the scientist, and are made available to create workflows. The tasks are generally coarse-grained, and can be very heterogeneous (unlike the tasks considered in the high performance computing community for instance). In the WaaS, the workflow topology must be given by the scientist in a YAML description file. Figure 1 presents the meta-grammar of the workflow description language, and Figure 2 presents an example of a workflow. In the WaaS the meta-grammar of the workflow description has to be implemented and customized by the Cloud provider to get its own specific grammar. This will be detailed in the second part of this section.

In the YAML description file, two main elements are required, the specification of *files*, and the specification of *tasks*. The `files` section is the list of all the files that are created during the workflow execution and transmitted from tasks to tasks. Each file is specified by an identifier (`id`), and a name (`name`). A file whose type is `output` (optional), is a file that the end-user wants to retrieve and analyze at the end of the execution. Similarly, the `tasks` section is the list of all the tasks of a workflow. To each task is associated the path and the name of the executable file to run (`app`), as well as the associated execution parameters (`params`), the hardware requirements of the task (`hardware`), the needed operating system with attached software dependencies (`os`), its input files (`input`) as well as its output files (`output`), as references to the `files` list. In `os`, the end-user has the

ability to specify a list of software dependencies that must be installed before launching the task. Available dependencies are customized by the Cloud provider (see the second part of this section), and all provisioning and installation operations are automated by the WaaS (by a mechanism close to a *Dockerfile*), thus making the resource management aspect transparent for the user.

One can note that for each file, or task, optional attributes can be added. Those optional attributes depends on the type of scheduler that the Cloud provider chooses to deploy, and on the type of hardware and operating systems available in their specific infrastructure. Thus, it is up to the Cloud provider to define the specific grammar of the description file accepted in its service, by defining the list of attributes required in the YAML description file. For example, it is up to the Cloud provider to define the list of `hardware` informations required or optional (*e.g.*, CPUs, RAM, GPU, etc.). Figure 2 presents an example of a workflow specification. The example file corresponds to the workflow depicted on the left-side of the figure. In this example, a few optional parameters have been added by the Cloud provider: the size of the files to transfer between tasks; the duration of each task as a number of instructions; the requirements in terms of CPUs and memory.




```
files:
    - id: File1
      name: inter.txt
      size: 1000 # Kbytes
    - id: File2
      name: out.txt
      size: 2096 # Kbytes
      type: output
tasks:
    - app: T1
      len: 22000
      needs:
          cpu : 1
          memory : 1024
      os: ubuntu18.04
      output :
          - File1
    - app: T2
      len: 34000
      needs:
          cpu: 2
          memory: 2048
      os: ubuntu18.04
      input:
          - File1
      output:
          - File2
      params: -i inter.txt -o out.txt
```

Fig. 2: Example of a workflow with two tasks by using a cloud-specific-grammar built from the meta-grammar.

Once a scientist has specified her workflow and has created its input files, she submits the workflow to the service. This submission is composed of the tasks (`app`), the input files (`input` of the entry tasks of the workflow), and the workflow description file in a single archive, and can be made on a simple web interface. Once a workflow has been correctly executed by the Cloud provider, its result files (output files) are available to the scientist via a web link. The scientist does not have any other work to do, thus it can be said that the execution is fully automatized and transparent for the end-user, (*i.e.*, as specified in Table I).

B. Cloud provider concerns and WaaS architecture

The entire execution of the workflow (*i.e.*, the scheduling, the resource management, the execution of the tasks as well as the file management) is handled by the WaaS service deployed on the Cloud provider side. The service is composed of two main modules, the Master and the Worker modules. The Master module is responsible for a cluster of Workers, where each worker is attached to one physical machine (*i.e.*, node). The bandwidth within a cluster is assumed to be homogeneous.

Master module - The Master module is hosted by one of the nodes of the infrastructure, and is accessible from the outside of the cluster in order to receive submission requests from end-users. It contains a scheduler sub-module. This scheduler is responsible for the following decisions: where (on which resource) and when to execute a task. In the WaaS the role of the scheduler is abstracted as a set of interfaces as follows. The scheduler provides for each task a slot $s =< t, b, e, r > \in \mathcal{S}$, where $t \in V$ is a task of a workflow $G = (V, A)$, where $b, e \in \mathbb{N}$ are the starting and ending times of the task, and $r \in \mathcal{R}$ is the virtual resource that will execute the task. Ending time e can be unset depending on the scheduler that is used, and the virtual resource r can be anything, depending on the virtualization technology that is used (even bare metal), but for the sake of simplicity we will use the term virtual resource. A virtual resource is defined by $r =< b, a, e, k, n >$, where $b, a, e \in \mathbb{N}$ are respectively the starting instant, the availability instant (after the boot process), and the ending instant (killing of the resource) of the virtual resource. The attribute $k \in \mathcal{K}$ represents the capacities of the VM, for instance vCPU, or memory. The attribute n is the node that will host the virtual resource. Depending on available information, the scheduler can determine the ending time e of a task, or may set an infinite lifetime for the task slot, and its associated virtual resource. In the rest of this paper, the assignment of the tasks to virtual resources through time (*i.e.*, the set of slots S) is called a planning. As long as the scheduler is designed in the respect of the above interfaces, any kind of scheduler can be used by the Master. The idea behind the modular scheduler is to provide a flexible solution for the Cloud providers, which may have different objectives in mind (*e.g.*, energy optimization, performance optimization, fairness, etc.). Note that different schedulers may need different kinds of information, thus explaining the meta-grammar of the workflow description file we presented in the previous subsection. Indeed, for example some schedulers may require different type of hardware information about the tasks, or average number of instructions of the tasks, as presented in the example in Figure 2.

The Master module execution pipeline (Figure 3) consists of two different parts: (1) calling the scheduler sub-module when a new workflow submission occurs to update the planning; (2) performing the execution of its current planning, by sending execution and provisioning orders to the Workers, and by managing new monitoring information from them. The types of orders that can be sent to the Workers by the Master module is presented in the next sub-part. The `wait event`

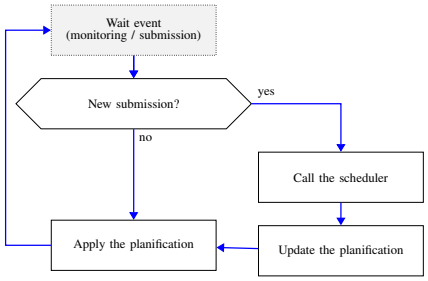


Fig. 3: Master execution pipeline

step of Figure 3 represents both monitoring and submission events, where monitoring events refers to the events sent by the Workers in response to orders.

```
name: value
hardware:
  # hardware capacities
  key1: value1
  ...
os:
  # OS and boot times
  name1: time1
  ...
*optional attributes :
```

Fig. 4: Meta-grammar of a worker description file

```
name: node_1
speed: 2200
hardware:
    cpus: 16
    memory: 16384
os:
    ubuntu: 30
```

Fig. 5: Example of description of a worker by using a cloud-specific-grammar built from the meta-grammar of Figure 4.

Worker module - One Worker module is deployed on each node of the Cloud infrastructure dedicated to scientific workflows. When deployed, a Worker module registers itself to the Master module responsible for its cluster. To the registration is attached information that can be customized by the Cloud provider, and that are described by a YAML file. The meta-grammar of this description file is presented in Figure 4, and an example is given in Figure 5 where one possible specification of this meta-grammar is presented. The `hardware` attributes defines the hardware capabilities of a node, and one can note the relationship with the hardware requirements of a task presented in Figure 1. The `os` section lists all the virtual resources that can be provisioned on the nodes, and the number of seconds it will require in average. One can note again the relationship between the `os` requirements of a task presented in Figure 1. The attributes of a node are typically used by the scheduler of the Master module and are reported to the grammar of the YAML file asked to the scientist.

A Worker module is responsible for launching and stopping the virtual resources, executing the tasks, and transferring the required files for tasks. However, the Worker module is not responsible for taking decisions, and simply answers to the orders sent by the Master module (*i.e.*, according to the planning created by the scheduler). There are four orders that can be received and addressed by a Worker:

- launch a virtual resource r;
- kill a virtual resource r;
- execute a task t on a virtual resource r;
- download a file f for a task t from a remote node n.

Note that the number of orders is small, and that only the first three depend on the type of virtualization. Therefore, defining new Workers to manage different types of virtual resources is quite straightforward by following the above interfaces. In addition, one can note that the virtual resources provisioned by the Worker module are customizable and may be stacked or unstacked, and may even run directly on the physical node (*i.e.*, bare metal), according to the specific choices of the Cloud provider. Finally, a cluster can contain heterogeneous Worker modules, *e.g.*, Workers providing container, along with workers providing VM. The idea behind this conceptual choice, is to enable the management and execution of heterogeneous workflows, that may require different kinds of operating systems to be properly executed. By putting the heterogeneity management at the Worker level, it becomes fully transparent for the end-users, unlike the solutions of the state of the art [3], which requires from them additional script files.

The last order of the Worker interface is to transfer files created by a task, and to ensure that the dependencies between the tasks are respected. To enable the transfer of the different files, each Worker module instance stores all the files produced during the execution of the workflows locally. When a Worker module requires access to a file that was created on another node, it will simply get it by sending a file download request to a sibling Worker module. In the workflow definition, multiple tasks could depend on the same file. For this reason, when multiple tasks are to be executed by the same Worker, and have the same file dependency, the file transfers are merged in order to prevent multiple copies of the files that would be useless. All the file transfers are managed by the Worker module automatically. For privacy reasons, virtual resources have access to the files needed by the tasks they are executing, but have no access to the other files (*i.e.*, the files of the other users). An additional small order can be noted on the Worker, the order to delete useless files (*i.e.*, files that are no longer required by any tasks, and that are not output files).

The types of operating systems that can be provisioned can be customized by deriving from a standard OS ("à la Dockerfile"), and all the dependencies are automatically installed by the Worker module, thus enhancing the heterogeneity management capabilities of our solution. Finally, virtual resources are provisioned only when required by the workers, and resources can be released when becoming useless, therefore, resource provisioning is elastic.

Federation of services - Until now we have described the part of the WaaS that manages the resources of a single cluster of nodes. Yet, most of the time the infrastructure offered by Cloud providers are composed of multiple clusters (assuming that a cluster refers to a single local area network). When handling multiple clusters, some issues are raised. For instance, some

scalability issues may be due to the high number of Workers handled by a single Master, or due to the high number of variables to consider in the scheduling problem. Furthermore, some issues can be raised by the network bandwidth bottlenecks between clusters etc.

In the current version of the WaaS, we address the problem due to the number of events that need to be handled by the Master module when considering multiple clusters of nodes. To address this explosion of events for the Master module, the intuition is to divide the planning execution between multiple Masters, hence reducing the number of events managed by a single Master (less Workers, tasks, files and virtual resources to manage). Another solution would have been to deploy as many WaaS services as there are clusters, and let the end-users choose on which cluster to execute their workflows. However, in case of big workflows or when clusters are highly loaded, a workflow can be poorly executed on a single cluster, while using multiple clusters could improve the execution quality. For this reason, we have opted for a *federated service* where each cluster is handled by one Master, and where a *Leader module* is deployed to federate the Master modules (Figure 6). In this federated version, the Leader module is the only one making decision and possessing a scheduler sub-module.

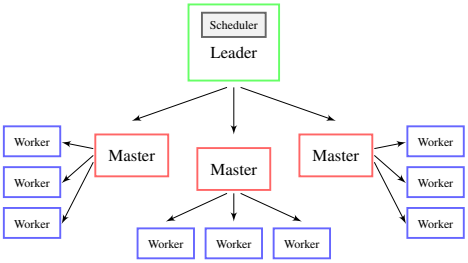


Fig. 6: WaaS deployed on 3 clusters with 3 nodes each

Thus, instead of calling the scheduler, each Master module receives a sub-planning to apply from the Leader module. This sub-planning received by a Master module only involves the nodes of the cluster under its supervision, as well as only the tasks that will be executed on its own cluster. Workers from different clusters cannot communicate, however the different Master modules can directly communicate to handle file transfers from point to point without the intervention of the Leader.

Note that since a distributed infrastructure with several clusters is targeted, decentralized scheduling algorithms would be needed, for scalability, in addition to federation. However, our results (Section IV) shows that our first scalability bottleneck is the number of events handled by the Master module, and not the scheduler resolution time. Furthermore we wanted the WaaS to be, as a first step, compatible with the centralized algorithms of the literature. Considering distributed scheduling algorithms could be the subject of future work.

To conclude this section, a new Cloud service for the execution of scientific workflows has been described. This service is highly customizable in terms of scheduling decisions, and resource provisioning, which enhances heterogeneity management and enables resource optimizations compared to the state of the art. Furthermore, the service is usable in a context of an infrastructure composed of multiple clusters, thus providing a good turnkey solution for Cloud providers.

IV. Evaluation

In this section, a detailed evaluation of the WaaS is presented. Each section refers to some of the objectives of Table II: (1) tackling heterogeneous workflows while not degrading the scientist concerns and the performance; (2) enabling specific optimizations from the Cloud provider perspective. The experimental protocol is detailed hereafter. All codes and results are available on a public *git* repository[2].

A. Experimental protocol

WaaS implementation - The WaaS has been implemented using the library Scala Akka. The architecture of the platform has been presented in Section III. The Scala Akka library is an actor oriented library. In the WaaS implementation, each module is an actor answering and sending messages to the other modules. To evaluate the capability of the WaaS to enable optimizations of resources for the Cloud provider, four different scheduling algorithms of the literature have been implemented, HEFT [6], Min-Min, Max-Min [7] and HEFT_*deadline* [8]. HEFT, Min-Min and Max-Min have originally been designed for the scheduling on Grid infrastructures, and have been adapted to be able to schedule virtual resources. This adaptation is presented for the HEFT algorithm in [8], and is similar for the Min-Min and Max-Min algorithms. Furthermore, to show that the WaaS tackles heterogeneity, two different Workers have been implemented, one providing KVM virtual machines, and one providing Docker containers. Recall that these workers are written at the level of the Cloud provider and that nothing is asked to the scientist. All these codes are available on the repository.

Execution infrastructure - Experiments have been run on a distributed infrastructure composed of two different clusters. These two clusters are part of the Grid'5000 [3] experimental platform, and are presented in Table III. The WaaS has been deployed on this infrastructure with five Master modules as depicted in Figure 7. One of them handles the whole *econome* cluster, and each of the four others handles a sub-cluster composed of 11 nodes of *ecotype*. In other words, *ecotype* is divided in four sub-clusters for experiments. On the one hand, the worker providing Docker virtualization has been instantiated and deployed on the nodes of two of the *ecotype* sub-clusters, and on the *econome* cluster. On the other hand, the worker providing KVM virtualization has been instantiated and deployed on the two remaining sub clusters of *ecotype*. The bandwidth between the *ecotype* and *econome* clusters is 10 Gbps. The Master modules are running on nodes that also host a Worker module, and the Leader module is running on a node that also hosts a Master module.

[2]https://github.com/EmileCadorel/WaaS
[3]https://www.grid5000.fr

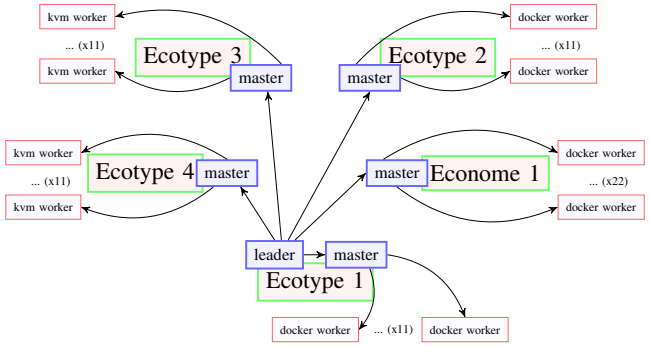


Fig. 7: Deployment of the WaaS on Grid'5000

Name	Nodes	CPU	Memory	Storage	Network
econome	22	Intel Xeon E5-2660 2.20GHz, 2 CPUs/node 8 cores/CPU	64 GB	2.0 TB HDD SATA	10Gbps
ecotype	44	Intel Xeon E5-2630L v4 1.80GHz, 2 CPUs/node 10 cores/CPU	128 GB	400 GB SSD	2 x 10Gbps

TABLE III: Hardware description of the nodes used in experiments.

Workload - The *Montage* workflow is a typical case-study used to evaluate workflow engines and scheduling algorithms [15], [18], [21], [22]. It is a real and complex workflow that integrates most of the workflow classes characterized by Bharati et al. in [23], and that creates astronomic mosaic images. The simulated workload is composed of 100 *Montage* workflows, each of them composed of 619 tasks. A total number of 400 executions of the Montage workflow has been performed, a hundred for each of the 4 different scheduling algorithms: HEFT, Min-Min, Max-Min and HEFT_deadline. Each workflow generates approximately 2.0 GB of files during their execution that have to be transferred to fulfill the tasks dependencies. Each workflow is submitted in a waiting queue and is processed by the Leader after the scheduling algorithm has planned the previous one, so approximately every 3 to 4 seconds. This waiting queue of submissions is required by the centralized scheduling algorithm. The average duration of the workload execution for the four execution is 24 minutes. To make comparison with the state of the art, the Montage workflow has also been written for the Argo workflow engine.

B. Results

In this section are presented the different results we obtained in relation with our objectives of Table II.

Scientist concerns - The submission of the Montage workflow composed of 619 tasks has been made 400 times in our experimental benchmark. As expected, the scientists (end-users) do not have to make any modification, even when different scheduling algorithms are deployed. The end-users also did not have to worry about the virtualization technologies that have been used, and chosen by the Cloud provider, and the resource management is fully transparent for the end-user. Note that the description file of the Montage workflow in our experiments has been generated automatically by the workflow generator provided by Pegasus, and was easily adapted for the WaaS with a transformation script. This file[4] is 11977 lines long. An almost similar description file is necessary for the state of the art workflow engines, such as Argo[5], thus we can state that properties of Table I are preserved by the WaaS in comparison to the state of the art. One can note from the git repository that the WaaS file is longer than the Argo file, this is due to two different reasons: (1) first in Argo the description of the workflow does not handle the dependencies between tasks through files specifications, but with abstract dependencies between the tasks (*i.e.*, edges), which reduces the specification size; (2) second Argo has template task descriptions, that we did not implemented in our prototype, but which could be easily added. Note that, by not handling dependencies between tasks with files, Argo is theoretically more generic but it requires from the users to manually handle files sharing through Kubernetes volumes. This reduces the transparency for the users. This will be further discussed later in this section.

Heterogeneity - Two different Workers have been deployed on the infrastructure, providing different types of virtualization. We do not intend to compare the different virtualization techniques, but to illustrate the ability of the WaaS to integrate and handle different Workers in a transparent manner for the scientist, thus handling heterogeneous workflows while preserving the scientists concerns. The code of the Workers can be found on the git repository[6], where only one file differs from KVM to Docker. This file is 200 lines long for Docker, and 500 for KVM. They both follow the interfaces defined in Section III and are written by engineers or operators at the level of the Cloud provider. Table IV lists the number of virtual resources provisioned by KVM Workers and Docker

[4]WaaS description of Montage
[5]Argo description of Montage
[6]Docker and KVM workers

Workers in our benchmark. VMs provisioned with KVM use Ubuntu 18.04, and each provisioning took 59 seconds in average, when the provisioning of a Docker container (also Ubuntu 18.04 container) took 3 seconds in average. Such transparent heterogeneity for the scientist is not possible in other workflow engines of the state of the art except Argo. But we will show that Argo degrades a lot the completion time of workflows to achieve heterogeneity. In Apache Airflow, tackling the heterogeneity of workflows is possible but it is up to the end-users to manage it. Indeed, to use heterogeneous resources, the end-users have to call different Cloud providers using configuration and script files, when it is completely transparent in the case of the WaaS.

Virtualization	HEFT	HEFT_deadline	Min-Min	Max-Min
Docker	4728	1197	4923	3981
KVM	1749	0	1539	1658

TABLE IV: Number of virtual resources provisioned during the experiment for each scheduler.

To understand how the management of heterogeneous workflow can be achieved with the WaaS, lets consider a simple workflow composed of two sequential tasks T1 and T2. Lets also assume that T1 needs to be executed on Windows, and that T2 needs to be executed on Ubuntu. On the one hand, if considering that clusters use Linux, there are no choice for the execution of T1: a VM has to be provisioned, thus the WaaS would consider either the cluster `Ecotype 3` or `Ecotype 4` (Figure 7). On the other hand, the task T2 can be executed on a container or a VM. Because containers have a faster boot process, they would probably be preferred (depending on the scheduler, and infrastructure load context), thus the clusters `Econome 1`, `Ecotype 1` or `Ecotype 2` would be selected for the execution of T2. In either case (execution of T1, and T2), the virtual resources are provisioned without any stacking of resources. In Argo the management of this same heterogeneous workflow would not be considered directly by Argo, but by the Kubernetes cluster that is selected to execute the pods requested by Argo. To run a container under Windows, there is no other choice than starting a Windows VM, and then start a container over it (which could be handled by Docker), thus creating a stack of virtual resources and a loss of performance. One can also note that in any case, Argo and Kubernetes cannot manage bare metal resources unlike the WaaS.

Even if the Montage workflow is not an heterogeneous workflow (every tasks has to be executed under Ubuntu), our evaluation shows that heterogeneous resources are used when the clusters that handle Docker containers are highly loaded. For example, with the HEFT algorithm, the 100th submitted workflow was executed by 29 different nodes. In these 29 nodes, 23 were running docker containers and 6 were running KVM virtual machines (Table IV). This proportion is of course not the same for all the workflows, according to the scheduling algorithm. For instance, HEFT chooses the virtual resource that provides the earliest finish time for the task, while taking into account the current load of the infrastructure.

Performance - The execution of the workflow Montage was made using both the WaaS and Argo. It is difficult to have a fair comparison between these two systems, as they operate in different contexts. To be as fair as possible, the comparison was performed on a single econome node, using Docker virtualization for WaaS, and a local Kubernetes for Argo (the deployment of Kubernetes is available on the source repository for reproductibility). Contrary to an execution with the WaaS, the full workflow Montage with 619 tasks cannot be executed on a single node by Argo, thus a smaller workflow was chosen composed of 31 tasks. We consider that this comparison is fair for two reasons. First, we execute a single workflow of a single user which is the kind of scenario handled by Argo which is not designed to manage multiple workflows of multiple users. Second, the use of only Docker containers in the WaaS ensures the same virtualization conditions as Argo (that uses Kubernetes).

The completion time of the workflow is 37 seconds for the WaaS, and 3 minutes and 32 seconds for Argo. This difference is due to the time taken by Kubernetes to starts new pods (1 per tasks), and configure them, when the WaaS only starts one virtual resource for the whole workflow (because only one user is involved). The time required to start a pod can be due to many parameters that we do not know with precision, therefore we will not develop further on this aspect. However, we are sure that this overhead is due to the pods as the execution times of the tasks are exactly the same for both the WaaS and Argo. Hence, even when virtual layers are not stacked and with a single workflow of 31 tasks, the overhead of Argo is significant.

From this experiment, some points should be discussed. First, Argo is not aware of the physical infrastructure and assumes that it has access to an infinite amount of resources. This is a major issue when using a small infrastructure and big workflows. On the contrary, the WaaS is able to take into account the small size of the infrastructure and adapt the execution to this context. This first point is further discussed in the following sub-part. Second, by running each task in a different pod, Argo loses a lot of time in resource provisioning. This second granularity issue is once again related to the unawareness of the physical infrastructure. Indeed, because the WaaS knows the capacities of each nodes, bigger virtual resources running multiple tasks can be used.

Optimizations - Furthermore, with our benchmark, we also aim at showing that different scheduling algorithms could be used or added by Cloud providers with minimal efforts. The code of these scheduling algorithms can be found on the git repository[7], where only one file of about 100 lines is required for each scheduler. Each scheduler has a different behavior and uses the infrastructure in a different way. We do not intend to compare the performances and results of the different algorithms. However, in order to validate the scheduler modularity, Figure 8 presents for each algorithm and for each submitted workflow (each 3 to 4 seconds) its ending

[7]Codes of the schedulers

time on the execution timeline of the experimental scenario. One can note that, as expected, each scheduler has its own behavior in response to the input workload.

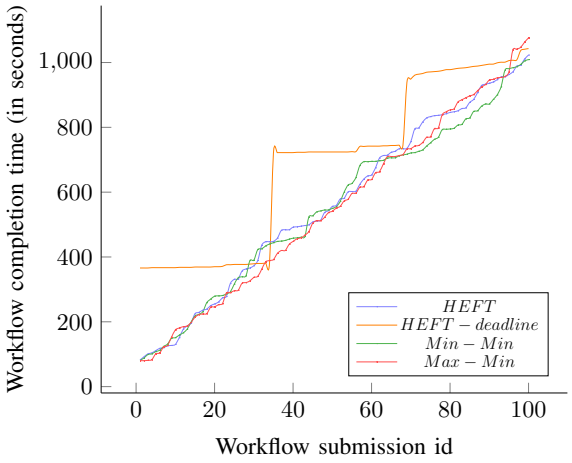


Fig. 8: Comparison of the behavior of each scheduler in response to the experimented workload. A workflow associated to an `id` is submitted each 3 or 4 seconds. The completion time of each workflow is depicted according to the scheduling algorithm to illustrate the modularity of the WaaS.

The deployed schedulers use the information about the infrastructure, and about the workflow topologies to take scheduling decisions. This is a completely different philosophy compared to other solutions as it enables the optimization of the resources at the level of the Cloud provider. Indeed, as already mentioned, in other workflow engines the management and the scheduling of the workflow is performed at the level of the scientist. As a result, the infrastructure is unknown by the embedded scheduling algorithm of the engine. For example, the scheduler of the Argo workflow engine simply starts a pod (Kubernetes virtual resource) for each task, as soon as they can be started. This can result in a high demand on the physical infrastructure. In our example with Montage 619 tasks, Argo would start 423 pods at the same time, a workload that cannot be handled by a single node, while the WaaS succeeds in running the workflow on a single node, by making decision based on the capacity of the infrastructure.

One can also note, that the solution provided by the WaaS service is elastic. For instance, with the HEFT algorithm, the execution of the first submitted workflow used 22 virtual resources with durations ranging from 11 seconds to 114 seconds with an average of 62 seconds. These virtual resources have executed 20 to 60 tasks each, with an average of 28 tasks. The completion time (*i.e.*, makespan) of this first workflow was 118 seconds. This shows that resources are created and deleted in an elastic manner, *i.e.*, with shorter lifetime than the overall makespan. Of course, the elasticity depends on the scheduling algorithm that is chosen. This is possible thanks to the Master-Worker and heterogeneous Workers architecture of our service.

Multiple cluster infrastructure - Our experimental benchmark also shows the ability of the WaaS service to use a distributed infrastructure composed of multiple clusters. Table V lists the number of tasks executed in each cluster. Every scheduler took approximately the same time to schedule one workflow, with an average of 2.5 seconds. Hence, the bottleneck of the execution is not located at the level of the scheduling algorithm, but on the number of messages that are transmitted from the Master to the Workers, as illustrated in Table VI by the number of messages processed by each Master module during the execution. The last line of this table additionally shows an approximation of the number of messages that would have been treated if only one Master module had been used. The distribution of the work between the different clusters depends on the decision taken by each scheduler, however, it may be noted that in all cases the workload is dispatched between multiple clusters. There are nothing to compare with the state of the art on that specific point as the management of the infrastructure is left to a third party when using existing workflow engines.

Cluster	HEFT	HEFT_deadline	Min-Min	Max-Min
ecotype-1	17095	21636	16752	16922
ecotype-2	10815	21777	11248	11244
ecotype-3	6051	0	5927	6196
ecotype-4	6434	0	6144	6342
econome-1	21505	18487	21829	21196

TABLE V: Distribution of the workload across the clusters, by the number of tasks executed

Cluster	HEFT	HEFT_deadline	Min-Min	Max-Min
ecotype-1	287402	211825	281331	287402
ecotype-2	104329	175991	112066	115575
ecotype-3	65892	18	64917	65692
ecotype-4	68613	17	65363	74851
econome-1	174047	143578	177889	169672
estimation all	570891	498031	571523	570716

TABLE VI: Distribution of the workload across the clusters, by the number of messages processed

Sharing of workflows - Finally, in this subpart we discuss the differences between Argo and the WaaS service in terms of sharing capacities between scientists, and execution reproducibility, which is an important aspect for the scientific community. First, it is important to precise that Argo has not been initially designed for the specific case of scientific workflows composed of tasks that share files, but for DAGs of tasks whose dependencies are symbolic relations. In other words, the management of the files between the tasks is not managed by Argo itself, but is left to the end-user who has to create a persistent volume in the Kubernetes cluster and place the input files inside it. This is an important drawback, first because there are a large variety of volume management systems in Kubernetes, and second because additional scripts have to be defined and shared by the end-users for reproducibility. Furthermore, scripts may have to be customized by each scientist. To our knowledge, all workflow engines targeting the FaaS services encounter this same issue. In the WaaS service the management of the files is directly tackled by the service, thus asking no additional work than the description

of the workflow to the end-users. Hence, even if the description of the workflow is a bit longer (files and file dependencies must be listed in the YAML description file), no additional scripts are required. Workflows are a single archive that can be shared between users with almost no need for customization. As explained in Section III, the customization of the workflow description file may be required if using two different Cloud providers that use different workers and schedulers. However, this customization is a descriptive customization while the customization in the state of the art would require system administration and scripting knowledge which are more difficult to acquire for scientists.

V. CONCLUSION

In this document has been presented a new Cloud service for the execution of scientific workflows, namely the Workflow-as-a-Service. Unlike the related work, the WaaS adopts a vision of a Cloud-provider-side service dedicated to scientific workflows that is presented as a turnkey solution to the Cloud providers. In this paper, we have shown that this conceptual approach of a workflow engine allows to handle highly heterogeneous workflows while enabling specific optimizations at the level of the Cloud provider, and while maintaining both the performance when executing workflows and the ease of use for the end-user (*i.e.*, the scientist). Indeed, by giving the responsibility of the scheduling of tasks and the resources provisioning to the Cloud provider, the Cloud provider collects more information than in other solutions, hence enabling important optimizations in the management of the infrastructure (energy optimization, fairness, performances, etc.). To facilitate the adoption of the WaaS by the Cloud providers and to improve the flexibility of the service, the WaaS has been made modular, thus facilitating the definition of new types of Workers (*i.e.*, virtualization), and the integration of new schedulers into the Master. Finally, the WaaS has been designed to be scalable, even when considering a complex distributed infrastructure with multiple clusters.

The service has been evaluated on a real distributed infrastructure divided in five clusters with four different scheduling algorithms, and two types of Workers (KVM and Docker). Consistency has been shown during the execution of hundreds of workflows. A comparison to Argo, a well-known FaaS-oriented workflow engine, has also been presented, and has shown improvements in terms of infrastructure optimizations and performance when handling heterogeneity.

In this paper, we have pointed out some elements that can be subject to future works. First, we plan to evaluate the service with dynamic scheduling algorithms, *i.e.*, dynamic change of decision [9], which is theoretically possible but needs to be validated by experimentation. Second, we plan to investigate the case of distributed scheduling algorithms, in order to remove all possible bottlenecks within the service. Finally, we plan to add some fault tolerance mechanisms to avoid the whole re-execution of workflows in case of failures.

REFERENCES

[1] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny, "Pegasus: Mapping scientific workflows onto the grid," in *Grid Computing*, M. D. Dikaiakos, Ed. Springer Berlin Heidelberg, 2004.

[2] L. Leslie, C. Sato, Y. Lee, Q. Jiang, and A. Zomaya, "Dewe: A framework for distributed elastic scientific workflow execution," *Conferences in Research and Practice in Information Technology Series*, vol. 163, 2015.

[3] "*Apache Airflow*. url: https://airflow.apache.org/."

[4] "*Argo Workflows*. url: https://argoproj.github.io/argo-workflows/."

[5] B. Balis, "Hyperflow: A model of computation, programming approach and enactment engine for complex distributed workflows," *Future Generation Computer Systems*, vol. 55, 2016.

[6] H. Topcuouglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, 2002.

[7] M. Maheswaran, S. Ali, H. Siegal, D. Hensgen, and R. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems," 1999.

[8] E. Cadorel, H. Coullon, and J.-M. Menaud, "A workflow scheduling deadline-based heuristic for energy optimization in Cloud," in *GreenCom 2019 - 15th IEEE International Conference on Green Computing and Communications*, 2019.

[9] ——, "Online Multi-User Workflow Scheduling Algorithm for Fairness and Energy Optimization," in *CCGrid2020 - 20th International Symposium on Cluster, Cloud and Internet Computing*, Melbourne, Australia, 2020.

[10] H. Arabnejad and J. Barbosa, "Fairness resource sharing for dynamic workflow scheduling on heterogeneous systems," in *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, 2012.

[11] R. Ferreira da Silva, T. Glatard, and F. Desprez, "Controlling fairness and task granularity in distributed, online, non-clairvoyant workflow executions," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 14, 2014.

[12] H. L. Röst and et al., "Openswath enables automated, targeted analysis of data-independent acquisition ms data," *Nature Biotechnology*, vol. 32, 2014.

[13] ——, "Tric: an automated alignment strategy for reproducible protein quantification in targeted proteomics," *Nature Methods*, vol. 13, 2016.

[14] L. Reiter, O. Rinner, P. Picotti, R. Hüttenhain, M. Beck, M.-Y. Brusniak, M. O. Hengartner, and R. Aebersold, "mprophet: automated data processing and statistical validation for large-scale srm experiments," *Nature Methods*, vol. 8, 2011.

[15] Q. Jiang, Y. Lee, and A. Zomaya, "Serverless execution of scientific workflows," 2017.

[16] S. Ristov, S. Pedratscher, and T. Fahringer, "Afcl: An abstract function choreography language for serverless workflow specification," *Future Generation Computer Systems*, vol. 114, pp. 368–382, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X20302648

[17] A. John, K. Ausmees, K. Muenzen, C. Kuhn, and A. Tan, "Sweep: Accelerating scientific research through scalable serverless workflows," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, ser. UCC '19 Companion. New York, NY, USA: Association for Computing Machinery, 2019, p. 43–50. [Online]. Available: https://doi.org/10.1145/3368235.3368839

[18] J. Kijak, P. Martyna, M. Pawlik, B. Balis, and M. Malawski, "Challenges for scheduling scientific workflows on cloud functions," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018.

[19] Y. Caniou, E. Caron, A. Kong Win Chang, and Y. Robert, "Budget-aware scheduling algorithms for scientific workflows on IaaS cloud platforms," Research Report RR-9088, 2017.

[20] H. Wu, X. Chen, X. Song, C. Zhang, and H. Guo, "Scheduling large-scale scientific workflow on virtual machines with different numbers of vcpus," *The Journal of Supercomputing*, Apr 2020. [Online]. Available: https://doi.org/10.1007/s11227-020-03273-3

[21] S. Abrishami, M. Naghibzadeh, and D. H. Epema, "Deadline-constrained workflow scheduling algorithms for infrastructure as a service clouds," *Future Generation Computer Systems*, vol. 29, 2013.

[22] L. F. Bittencourt and E. R. M. Madeira, "Hcoc: a cost optimization algorithm for workflow scheduling in hybrid clouds," *Journal of Internet Services and Applications*, vol. 2, 2011.

[23] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M. H. Su, and K. Vahi, "Characterization of scientific workflows," in *2008 Third Workshop on Workflows in Support of Large-Scale Science*, 2008.