Reproducibility of computational experiments on Kubernetes-managed container clouds with HyperFlow

Michał Orzechowski[1], Bartosz Baliś[1], Renata G. Słota[1], and Jacek Kitowski[1,2]

[1]AGH University of Science and Technology, Department of Computer Science, Krakow, Poland
[2]AGH University of Science and Technology, ACK Cyfronet AGH, Krakow, Poland
{morzech, balis, rena, kito}@agh.edu.pl

Abstract. We propose a comprehensive solution for reproducibility of scientific workflows. We focus particularly on Kubernetes-managed container clouds, increasingly important in scientific computing. Our solution addresses conservation of the scientific procedure, scientific data, execution environment and experiment deployment, while using standard tools in order to avoid maintainability issues that can obstruct reproducibility. We introduce an Experiment Digital Object (EDO), a record published in an open science repository that contains artifacts required to reproduce an experiment. We demonstrate a variety of reproducibility scenarios including experiment repetition (same experiment and conditions), replication (same experiment, different conditions), and propose a smart reuse scenario in which a previous experiment is partially replayed and partially re-executed. The approach is implemented in the HyperFlow workflow management system and experimentally evaluated using a genomic scientific workflow. The experiment is published as an EDO record on the Zenodo platform.

Keywords: scientific workflows, reproducibility, cloud computing, application containers, container clouds, Kubernetes

1 Introduction

Reproducibility, a fundamental quality of the experimental scientific method, requires conservation of three basic component of scientific experiments [15]: *scientific procedure* describing the steps of the experiment, *scientific data* required as input of the experiment and produced as its results, and *scientific equipment* necessary to conduct the experiment. In the case of computational sciences, the scientific procedure can be described as a scientific workflow [6], input data are typically files, while the equipment is the execution environment of the experiment.

Application containers are often viewed as a means to scientific workflow reproducibility [3, 17], but containers alone solve only part of the problem enabling conservation of individual software components. Some approaches tackle

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_16

reproducibility of the broader execution environment, but propose non-standard solutions that suffer from lack of maintainability [16], and do not support reproducibility of complex experiment deployments. In this paper, we discuss modern *container clouds* as a comprehensive solution for reproducibility of computational experiments. We propose a solution for reproducibility of scientific workflows, focusing on Kubernetes-managed container clouds. The main contributions of this paper are as follows:

- We provide a solution that supports conservation of *in silico* experiments at various levels: the scientific procedure, the scientific data, the execution environment, and the experiment deployment.
- The solution covers a variety of reproducibility scenarios, including repetition, replication, and *smart reuse* of in silico experiments.
- We introduce an *Experiment Digital Object*, a record that captures information required to reproduce a scientific workflow experiment.
- We study the implementation of the solution in the context of Kubernetes-managed container clouds, an increasingly important computing infrastructure for scientific computing.
- We show an evaluation of the solution by studying reproducibility scenarios of a genomic scientific workflow. The experimental part is published as an Experiment Digital Object on the Zenodo platform as a demonstration of the solution.

In the paper, we adopt the following terminology for different types of reproducibility (partially adopted from [5]):

- **repetition**: the experiment is reproduced in exactly the same conditions, including the workflow specification, input data and the execution environment (OS, libs, even machine types and cluster configuration).
- **replication**: the same experiment is reproduced but in a different execution environment.
- **(smart) reuse**: intermediate results from a previous experiment are partially reused (without re-computing them), whereas the rest is re-executed due to changes, e.g. a new version of the scientific software.
- **reproduction**: broad term denoting any of the above cases.

The paper is organized as follows. Section 2 surveys current work on reproducibility of scientific workflows. Section 3 describes the proposed reproducibility solution, while section 4 contains its experimental evaluation. Section 5 concludes the paper.

2 Related Work

Container clouds are increasingly adopted as an infrastructure for computational experiments [4, 9]. A container cloud adds an additional layer on top of virtual machines, in which clusters of nodes are formed from VMs and are managed by

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_16

a *container management platform*. Container managers such as Mesos, Docker Swarm and Kubernetes have been studied in the context of scientific workflow management [11,18]. However, while many papers focus on application containers as a tool for achieving reproducibility of computational experiments [3, 17], containers alone address only the conservation of the runtime environment of individual software components, not the entire execution environment.

Several approaches employ logical conservation of the target execution environment for scientific workflows. In [16], a custom OWL ontology for describing the computational infrastructure used in a computational experiment is proposed. In [14], TOSCA specifications of the underlying infrastructure are used to deploy workflows on different cloud platforms in a portable way. The problem with such proprietary solutions that focus on portability and abstraction [1, 12] is their maintainability, a very important quality crucial for reproducibility. Given the complexity and dynamic evolution of computing infrastructures, it is difficult to maintain in-house provisioning and configuration management tools dedicated for scientific computing. In our experience, even industry-leading tools, such as Terraform, sometimes have this problem and cannot be treated as a universal solution in every situation. Finally, existing approaches do not address reproducibility of the entire complex *experiment deployment*.

Our approach supports conservation of the experiment procedure, data, execution environment, and experiment deployment. First, using Docker containers ensures conservation of the operating system and software packages. For provisioning of the computing infrastructure (virtual machines, storage resources), we rely on standard tools that fit best the particular computing environment and satisfy infrastructure reproducibility requirements, such as Terraform, Kubespray, or native tools of the cloud provider. Finally, we argue that configuration management of individual components and nodes is not enough. Complex experiment deployments – orchestration of component startup, dynamic provisioning of persistent volumes, replication and fault tolerance strategies, etc. – also need to be described in a reproducible way. That is where Kubernetes comes into play as a universal application deployment and execution platform. We chose Kubernetes [13] because it is supported by virtually all major cloud providers (Amazon EKS, Azure AKS, Google GKE) and because it is portable and universal – complex deployments can be reproduced using the same Kubernetes description files on various infrastructures. Kubernetes is also increasingly used in scientific computing [4]. The advantage of this approach is not only high reproducibility, but also portability and maintainability.

3 Experiment reproducibility with HyperFlow

This section describes the overall methodology for reproducibility of scientific workflows on Kubernetes-managed container clouds using the HyperFlow workflow management system. Details regarding reproducibility capabilities of HyperFlow on the level of workflow description, workflow execution and workflow deployment are also described.

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_16

3.1 Experiment digital object record

Fig. 1 presents an overview of the *Experiment Digital Object* (EDO) record, a collection of data and executable files that contain information on all software and data artifacts involved in the experiment, execution traces, and scripts for their basic analysis. This record is created after the experiment and can be published on an Open Science Platform. We have chosen Zenodo because of its relative popularity. Zenodo allocates a unique DOI identifier to the record which can be used in references. Records published in Zenodo are immutable which is important for reproducibility. Changes can be made and published as a new version of the record.

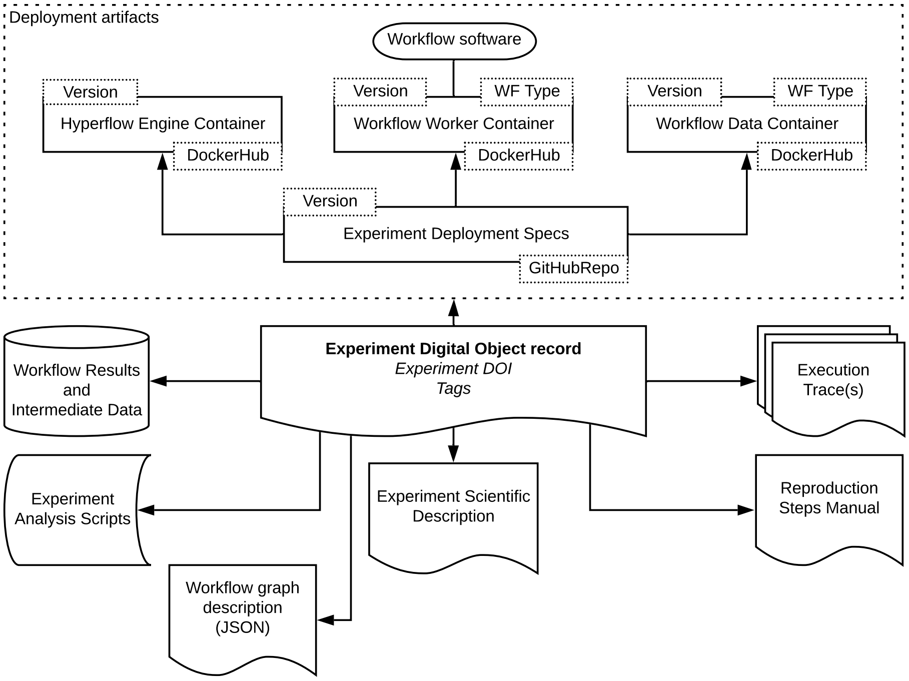
Deployment artifacts
Workflow software
Version
Hyperflow Engine Container
DockerHub
Version
WF Type
Workflow Worker Container
DockerHub
Version
WF Type
Workflow Data Container
DockerHub
Version
Experiment Deployment Specs
GitHubRepo
Workflow Results and Intermediate Data
Experiment Digital Object record
Experiment DOI
Tags
Execution Trace(s)
Experiment Analysis Scripts
Experiment Scientific Description
Reproduction Steps Manual
Workflow graph description (JSON)

Fig. 1. Experiment Digital Object (EDO) record contains all information required to reproduce a computational experiment managed by the HyperFlow WMS.

An EDO record is an entry point to experiment reproduction. It describes specific steps required to reproduce the experiment, but also points to specific versions of all deployment artifacts involved in this particular experiment. It also contains execution traces collected during the experiment run, for convenience converted to a CSV format. Experiment analysis scripts are also provided that process the execution trace as a data frame and produce useful charts.

DOI: 10.1007/978-3-030-50371-0_16

3.2 Workflow description language

HyperFlow introduces a simple workflow description language with strong semantics [2]. HyperFlow has been shown to successfully convert and run workflows from such system as Pegasus [7]. Such workflow interoperability is an important aspect of reproducibility. Listing 1.1 shows a fragment of a HyperFlow workflow description. The description contains mainly two sections: an array of *processes* (workflow tasks), and an array of *signals* (inputs and outputs of tasks). The listing shows an example process entry in which the *name* identifies the type of processing performed by the task, while *config* contains information passed to the executor on a remote node in order to run the task. The signal entry describes a file which is one of the inputs of the shown task.

Listing 1.1. HyperFlow workflow description (fragment).

```
 1  processes: [ {
 2    "name": "alignment_to_reference",
 3    "function": "k8sCommand",
 4    "config": {
 5      "executor": {
 6        "executable": "bwa-wrapper",
 7        "args": [ "mem", "-t", "2", "-M",
 8          "Gmax_275_v2.0.fa",
 9          "USB-001_1.fastq",
10          "USB-001_2.fastq"
11        ],
12        "cpuRequest": "1",
13        "memRequest": "500M",
14        "stdout": "20180321-083514-USB-001_aligned_reads.sam"
15      }
16    },
17    "ins": [1,2,3,4,5,6,7,8,9,10,11,12],
18    "outs": [13]
19  }, ... ],
20  signals: [ {
21    "name": "Gmax_275_v2.0.fa",
22    "type": "file",
23    "size": "990744229",
24    "md5sum": "3aa6cf1962f5260cf1405e82efb25c71"
25  }, ... ]
```

Such workflow representation is flexible and annotations can be easily (manually or automatically) added to the workflow description without interfering with other parts of the description. Several such annotations are shown in this example:

- CPU and memory requests (lines 12-13) are inserted by an execution planner (on the basis of historical execution traces or a prediction model) and are used by the underlying Kubernetes scheduler to optimize the placement of workflow jobs. During experiment replication these values can be changed to

perform controlled experiments regarding observation of their effect, e.g., on the execution time.
- Detailed information about file size and its md5 hash (lines 23-24) are useful during experiment reproduction to verify if the file has not been changed or corrupted.

3.3 Workflow execution

Workflow execution traces. During workflow execution, HyperFlow collects various information, such as job execution events (creation, pending, start and completion), resource usage metrics (time series) and provenance records. This information is converted to data frames and saved as CSV files which are included in the Experiment Digital Object record, along with the description of columns. Such an approach is quite versatile. Data from many experiments can be analyzed in a data frame parallel processing framework (e.g. Pyspark), or imported into a chosen database. Addition of new columns preserves backward compatibility, provided that the format and semantics of ”old” columns remain unchanged. Simple data analysis tools (python scripts) are included in the EDO record for quick and simple presentation of the experiment.

Workflow persistence and smart reuse. HyperFlow provides mechanisms for persistence and recovery of workflow executions using the *event sourcing* approach. In event sourcing, all changes made to the system during the execution are recorded as *events*. To recover the state of the system, events are replayed and the state changes are applied again but without repeating the side effects of the original operations. In the case of scientific workflows, the record of execution events contains information about completed job executions along with inputs consumed and outputs produced by them. The side effects not repeated during the recovery are actual running of the jobs and creation of their output files.

HyperFlow supports advanced *smart reuse* scenarios in which a previously recorded experiment is partially replayed (by reusing intermediate data from the experiment record), while other parts of it have been changed and need to be re-executed. To support smart reuse, intermediate data produced during the experiment must be persisted. Smart reuse scenarios include the following cases:

- Some input data files have been updated. Consequently, tasks consuming these inputs and all their *successors* (dependent tasks) need to be re-executed.
- Part of the workflow software has been upgraded to a new version. Tasks using this software and all their *successors* need re-execution in this case.
- Some intermediate data has been corrupted, so it cannot be reused during experiment reproduction. In this case, the intermediate data needs to be re-created, but only if it is needed by the parts of the workflow that must be re-executed. This can lead to a cascade re-execution of some or all *predecessor* tasks of the affected task that produces the intermediate data in question.

HyperFlow provides a tool called `hflow-recovery-analysis` which annotates an existing experiment execution records with information on which tasks need re-execution based on a configuration file consisting of the following entries:

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_16

```
changed: {
  "selector": <spec>,
  "value": "<value>"
}
```

Each entry specifies one or more workflow objects (processes or data) that have been changed. For example `{"selector":"process.name","value":"foo"}` selects all processes whose *name* is *foo*. Consequently all tasks invoked by such processes and their successors will be re-executed. Since the selector may apply to any attribute found in the workflow description and can address groups of objects, this mechanism is flexible and effective even for very large workflow graphs. Moreover, the tool can also be pointed to a directory where workflow intermediate data is located to check if it is not missing or corrupt. If this is the case, additional tasks can be be marked for re-execution.

3.4 Workflow deployment and execution environment

The final aspect of reproducibility concerns the computing environment and the deployment of the experiment in this environment. Let us describe this aspect at different levels in the context of Kubernetes-managed container clouds.

Computing infrastructure. The computing infrastructure level comprises Virtual Machines, storage resources, and the Kubernetes cluster. There are several options to address reproducibility at this layer, depending on the target infrastructure. In fact, we argue that choosing the right tool in the right situation is crucial.

- Terraform[1] is a widely used Infrastructure-as-Code tool designed to provision reproducible infrastructures, and supporting a large number of providers, including all major clouds.[2]
- Kubespray a tool that automates deployment of ”bare-metal” Kubernetes clusters (that includes deployments on IaaS clouds). [3]
- Native clients of a particular infrastructure. This approach can be very effective for managed Kubernetes clusters offered in the PaaS model, such as Google GKE, Amazon EKS, or Azure AKS.

In the experiments shown in this paper we use the third option as it is sufficient and effective.

OS, libs and application software. This layer is where application containers come into play. We use Docker containers for the software involved in the workflow execution: HyperFlow engine, Redis server, workflow workers, and – in the particular deployment used in this paper – the NFS server. Versioning of

[1] https://www.terraform.io
[2] https://www.terraform.io/docs/providers/type/major-index.html
[3] https://github.com/kubernetes-sigs/kubespray

container images is crucial for reproducibility. We employ a Continuous Integration / Deployment (CI/CD) pipeline which automatically builds and publishes new container images whenever a tagged commit is pushed into the appropriate repository.

Workflow data. We support conservation of workflow data by employing *data containers*. A data container contains input files of the workflow or, alternatively, provisions them before the execution. Data containers with specific input data sets can be prepared, versioned and published. They can be generic and used in a family of similar workflows, or very specific, created for a particular workflow, and even contain the workflow graph description file. The image of the workflow data container contains metadata information regarding such details. Moreover, data containers can be created to conserve output and intermediate data of a particular experiment run.

Experiment deployment. A computational experiment is not only a collection of software and data artifacts, but rather their concrete, complex deployment which also must be reproducible. For this purpose, we use the *Kubernetes YAML manifests* which allow reproducible, declarative deployment approach. The structure of the deployment used in the experimental evaluation is depicted in Fig. 2. The *HyperFlow engine* container runs the workflow described in the `workflow.json` file and creates *Workflow worker containers* running workflow jobs. A job container runs a *HyperFlow job executor* which communicates with the HyperFlow engine through a *Redis service* running in a separate container. Workflow data is shared through a *Persistent volume* which uses an *NFS server* running in another container. The NFS server is populated with workflow data via the *Workflow data container*. A deployment contains many important configuration parameters that needs to be preserved, e.g. environment variables that influence the behavior of individual components.

Listing 1.2. Example customization of the experiment deployment configuration.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hyperflow-engine
spec:
  template:
    spec:
      containers:
      - name: hyperflow
        env:
        - name: HF_VAR_WORKER_CONTAINER
          value: "hyperflowwms/soykb-workflow-worker:v1.0.6"
```

To version the deployment, we simply point to a particular commit in the github repository and use Kustomize[4] to configure the deployment with specific

[4] https://github.com/kubernetes-sigs/kustomize

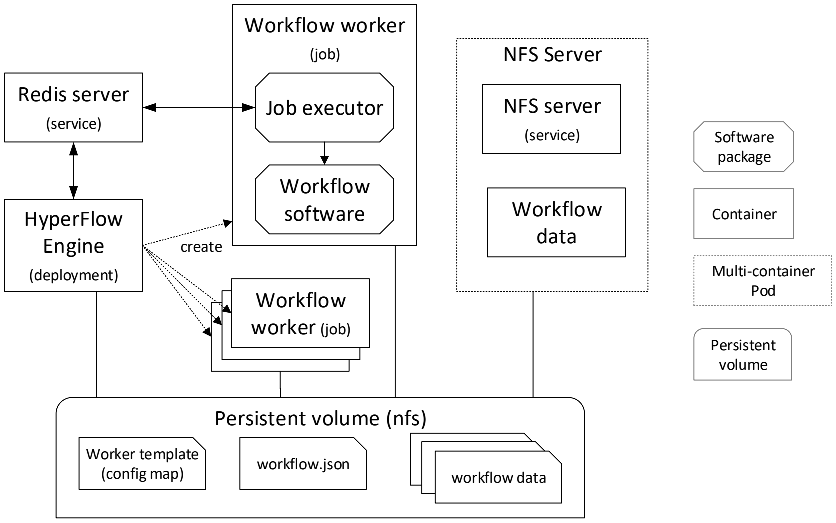

Fig. 2. Scientific workflow deployment used in the reproduction experiments.

versions of the containers. Kustomize allows one to create declarative customizations of the base Kubernetes YAML manifests. A customization is simply one or more YAML files that change (patch) specific parts of their counterparts from the chosen base. In our case, the customization captures specific parameters of the deployment of the original experiment, mainly the containers and their versions, as shown in the example on Listing 1.2. A customization is also a convenient place to modify these parameters and reproduce the experiment in different conditions, e.g. using new software versions. Finally, a customization can be published as an artifact documenting the experiment conditions and enabling its reproducibility.

4 Experimental evaluation

4.1 Methodology, experiment setup and original run

In the experiments, we have used the SoyKB genomic workflow [10]. Four experiments have been conducted:

- The *original run* of the workflow on a Google Kubernetes Engine with four nodes, each having 2 virtual CPUs and 1.93 GB memory. This run produced the Experiment Digital Object record along with other artifacts.
- The *repetition run* in which the experiment was repeated exactly in the same conditions.
- The *replication run* wherein the experiment was replicated on our in-house Kubernetes cluster deployed at the Cyfronet Computing Centre. For this experiment, we set up a 4-node cluster, where each node had 6 vCPUs and 22GB of memory.

- The *reuse run* wherein the smart reuse capabilities of HyperFlow were demonstrated using the GKE cluster.

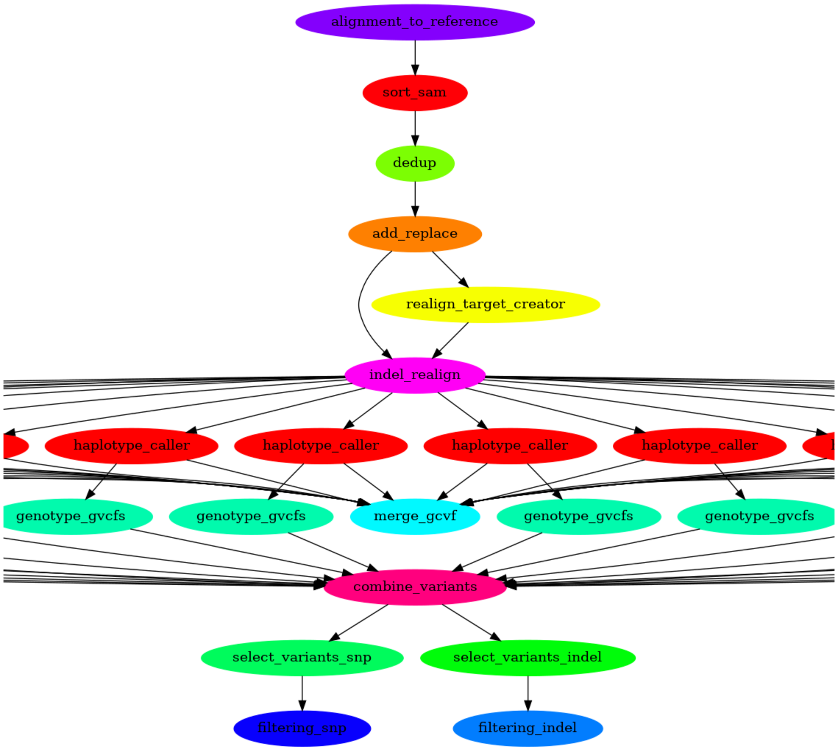

Fig. 3. Visualization of the experimental SoyKB workflow (simplified).

The experimental workflow consists of 52 tasks, as depicted in Fig. 3, and requires 2.6 GB of input data. The EDO record of the original experiment run was assigned a DOI and was published on Zenodo[5]. The execution of the original run is visualized in Fig. 4 (top). The chart, produced from the execution traces using a python script (published as part of the EDO record), shows the execution time of all workflow tasks, type of tasks (denoted by different colors), and their mapping to nodes whose names are indicated on the y axis. Note that the same node occurs multiple times, denoted as *nodeName-n*, if tasks run in parallel on this node. In this case, all tasks were configured with a *cpuRequest* of 0.5 vCPU. Since each node provided 1.93 vCPU and about 0.5 of it was reserved by the middleware, only two parallel tasks could fit in a node without exceeding the 1.93 vCPU supply, which is visible on the graph. We have tried to lower the vCPU

[5] https://doi.org/10.5281/zenodo.3659211

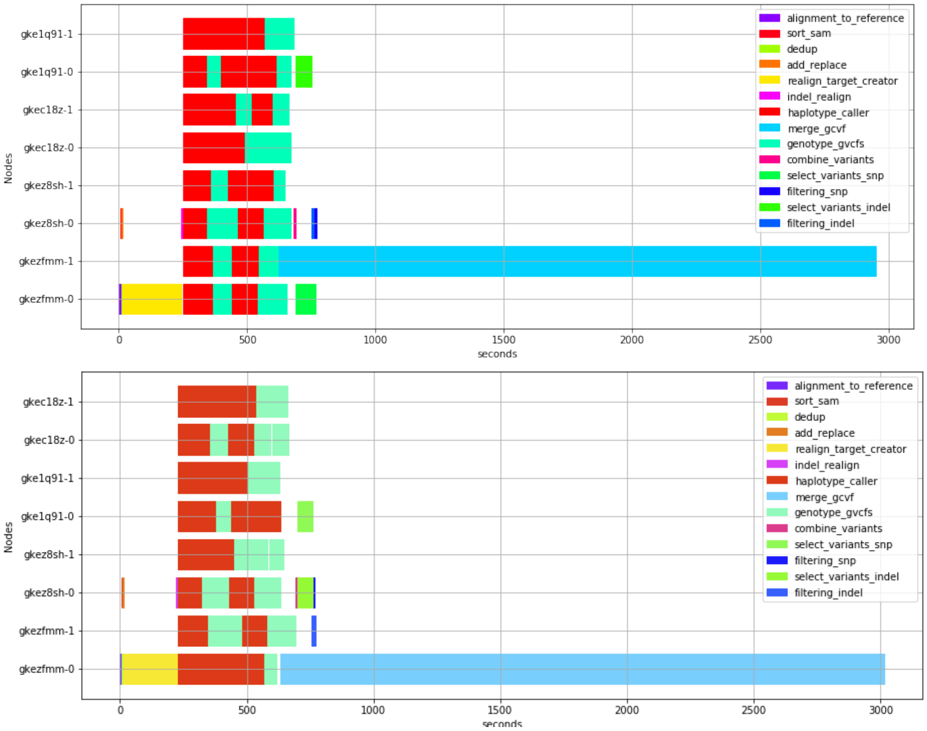

Fig. 4. Visualization of the original and repeated run of the SoyKB workflow on a 4-node Google Kubernetes Engine cluster.

requests so that more tasks would fit a node, but it resulted in job evictions due to out of memory errors.

4.2 Experiment repetition and replication

Fig. 4 (bottom) shows a visualization of the repetition of the experiment in the same infrastructure, while Fig. 5 shows its replicated execution on the Kubernetes cluster in the Cyfronet Computing Centre.

The repeated run is very similar, even the general mapping of tasks to nodes is much the same, albeit with some differences. This hints that the execution environment behaves in a relatively predictable way despite its complexity. This was proven by more runs which yielded similar results. The replicated experiment runs on much faster nodes, so all concurrent workflow tasks could run in parallel in the cluster. The resulting makespan is therefore significantly better.

4.3 Smart experiment reuse

The final experiment demonstrates the smart reuse scenario. This scenario is made possible by HyperFlow's capability to persist its execution state. In compliance

DOI: 10.1007/978-3-030-50371-0_16

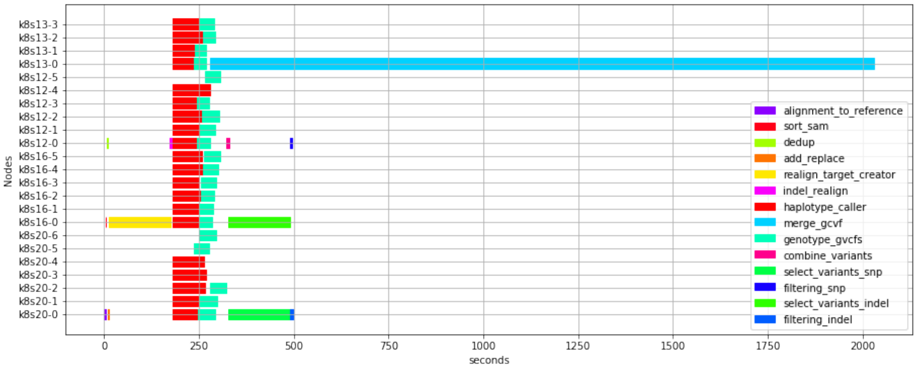

Fig. 5. Visualization of the replicated experiment run on Cyfronet Kubernetes cluster.

with the event sourcing model, this is done in the following steps: (1) run a workflow job to completion; here, job represents an operation that changes the workflow execution state but also causes side effects (writes files); (2) persist an event `job completed` in an *execution journal* file; (3) update the internal state of the workflow engine. The execution journal file can be used to *replay* a previous workflow execution, either in part or in full. A common scenario is fault tolerance where the workflow is restarted from a point its execution failed, after the cause of the failure has been resolved. The smart reuse scenario involves a full replay of the workflow, but some its parts are marked as changed and these are forced to be re-executed. To this end, the user prepares an appropriate configuration file which in our case looks as follows:

```
[ changed: {
  "selector": "process.name",
  "value": "genotype_gvcfs"
} ]
```

This configuration indicates that all workflow tasks whose name is `genotype_gvcfs` have been changed. The nature of the change can vary, e.g. it can denote the upgrade of the workflow software used by the tasks. The user runs the `hflow-recovery-analysis` tool which annotates the execution journal file marking all `genotype_gvcfs` tasks and their successors (task subtree) for re-execution.

The visualization of this execution is shown in Fig. 6. As one can see, only the last 25 tasks of the workflow have been repeated. It must be noted that the workflow can be re-executed in such a way from an arbitrary point only as long as the side effects of the original run – i.e. the intermediate data – are persisted and can be reused. The annotation tool can optionally verify if this is the case and, if some intermediate data are missing, analyze the execution graph and mark additional predecessor tasks for re-execution. In our case, we simply reused the storage volume of the original run which contained the intermediate data, but in a production setting a more robust storage solution would be required. Also,

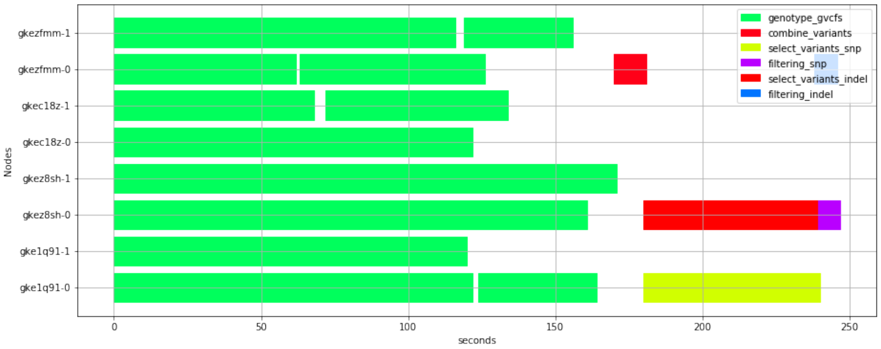

Fig. 6. Visualization of the experiment reuse scenario.

a trade-off should be considered whether it is more cost-effective to re-execute a task and generate its intermediate data again, or persist the data for future reuse. This is, however, out of scope of this paper.

5 Conclusions and Future Work

The reproducibility of the computational experiments is important not only for the domain scientists, but also very useful for workflow research, e.g. for conducting controlled experiments in the area of distributed computing management. We have shown a solution which facilitates repetition and replication of *in silico* experiments, focusing on scientific workflows and container cloud infrastructures managed by Kubernetes. Our solution supports conservation of not only the scientific procedure, but also the execution environment and the experiment deployment. We have also proposed a smart reuse scenario which supports partial reuse of previous experimental runs while configuring which parts have changed and need re-execution. An Experiment Digital Object that we have introduced contributes to open science and the idea of so called executable papers whose results are fully reproducible. Future work involves enrichment of the EDO record with more useful data and software for presentation of the experiment, and integration with a more robust storage solution combined with transparent data access using the Onedata platform [8].

Acknowledgements. The research presented in this paper has been partially supported by the funds of Polish Ministry of Science and Higher Education assigned to the AGH University of Science and Technology.

References

1. Azarnoosh, S., Rynge, M., Juve, G., Deelman, E., Niec, M., Malawski, M., Da Silva, R.F.: Introducing precip: an API for managing repeatable experiments in the cloud. In: 2013 IEEE 5th international conference on cloud computing technology and science (CloudCom). pp. 19–26. IEEE (2013)

2. Balis, B.: Hyperflow: A model of computation, programming approach and enactment engine for complex distributed workflows. Future Generation Computer Systems 55, 147 – 162 (2016)
3. Bartusch, F., Hanussek, M., Kruger, J., Kohlbacher, O.: Reproducible scientific workflows for high performance and cloud computing. In: 19th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Computing (CCGRID). pp. 161–164 (2019)
4. Beltre, A.M., Saha, P., Govindaraju, M., Younge, A., Grant, R.E.: Enabling hpc workloads on cloud infrastructure using kubernetes container orchestration mechanisms. In: IEEE/ACM Int. Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC. pp. 11–20. IEEE (2019)
5. Cohen-Boulakia, S., Belhajjame, K., et al.: Scientific workflows for computational reproducibility in the life sciences: Status, challenges and opportunities. Future Generation Computer Systems 75, 284–298 (2017)
6. Deelman, E., Gannon, D., Shields, M., Taylor, I.: Workflows and e-science: An overview of workflow system features and capabilities. Future Generation Computer Systems 25(5), 528–540 (2009)
7. Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P.J., Mayani, R., Chen, W., da Silva, R.F., Livny, M., et al.: Pegasus, a workflow management system for science automation. Future Generation Computer Systems (2014)
8. Łukasz Dutka, Wrzeszcz, M., Lichoń, T., Słota, R., Zemek, K., Trzepla, K., Łukasz Opioła, Słota, R., Kitowski, J.: Onedata – a step forward towards globalization of data access for computing infrastructures. Procedia Computer Science 51, 2843 – 2847 (2015), international Conference On Computational Science, ICCS 2015
9. Herbein, S., Dusia, A., Landwehr, A., McDaniel, S., Monsalve, J., Yang, Y., Seelam, S.R., Taufer, M.: Resource management for running hpc applications in container clouds. In: Int. Conf. on High Perf. Computing. pp. 261–278. Springer (2016)
10. Joshi, T., Valliyodan, B., Khan, et al.: Next generation resequencing of soybean germplasm for trait discovery on xsede using pegasus workflows and iplant infrastructure (2014)
11. Liu, K., Aida, K., Yokoyama, S., Masatani, Y.: Flexible container-based computing platform on cloud for scientific workflows. In: 2016 International Conference on Cloud Computing Research and Innovations (ICCCRI). pp. 56–63. IEEE (2016)
12. Nguyen Minh, B., Tran, V., Hluchy, L.: Abstraction layer for development and deployment of cloud services. Computer Science 13, 79–88 (2012)
13. Orzechowski, M., Balis, B., Pawlik, K., Pawlik, M., Malawski, M.: Transparent deployment of scientific workflows across clouds-kubernetes approach. In: IEEE/ACM Int. Conf. on Utility and Cloud Computing Companion. pp. 9–10. IEEE (2018)
14. Qasha, R., Cała, J., Watson, P.: A framework for scientific workflow reproducibility in the cloud. In: 2016 IEEE 12th International Conference on e-Science (e-Science). pp. 81–90. IEEE (2016)
15. Santana-Perez, I., Pérez-Hernández, M.S.: Towards reproducibility in scientific workflows: An infrastructure-based approach. Scientific Programming 2015 (2015)
16. Santana-Perez, I., da Silva, R.F., Rynge, M., Deelman, E., Pérez-Hernández, M.S., Corcho, O.: Reproducibility of execution environments in computational science using semantics and clouds. Fut. Generation Computer Systems 67, 354–367 (2017)
17. Stubbs, J., Talley, S., Moreira, W., Dooley, R., Stapleton, A.: Endofday: A container workflow engine for scalable, reproducible computation. In: Proceedings of the 8th International Workshop on Science Gateways, IWSG (2016)
18. Zheng, C., Tovar, B., Thain, D.: Deploying high throughput scientific workflows on container schedulers with makeflow and mesos. In: 17th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Computing (CCGRID). pp. 130–139. IEEE (2017)

ICCS Camera Ready Version 2020
To cite this paper please use the final published version:
DOI: 10.1007/978-3-030-50371-0_16