WfCommons: Data Collection and Runtime Experiments using Multiple Workflow Systems

Henri Casanova*, Kyle Berney*, Serge Chastel, Rafael Ferreira da Silva‡
*Information and Computer Sciences Department, University of Hawaii, Honolulu, HI, USA
‡National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, TN, USA
{henric,berneyk}@hawaii.edu, schastel.at.work@gmail.com, silvarf@ornl.gov

***Abstract*—Scientific workflows have become ubiquitous across scientific fields, and their execution methods and systems continue to be the subject of research and development. Most experimental evaluations of these workflows rely on workflow instances, which can be either real-world or synthetic, to ensure relevance to current application domains or explore hypothetical/future scenarios. The WfCommons project addresses this need by providing data and tools to support such evaluations. In this paper, we present an overview of WfCommons and describe two recent developments. Firstly, we introduce a workflow execution "tracer" for Nextflow, which significantly enhances the set of real-world instances available in WfCommons. Secondly, we describe a workflow instance "translator" that enables the execution of any real-world or synthetic WfCommons workflow instance using Dask. Our contributions aim to provide researchers and practitioners with more comprehensive resources for evaluating scientific workflows.**

***Index Terms*—Scientific workflows, workflow instance collection, workflow instance execution**

I. Introduction

Scientific workflows are relied upon by thousands of researchers for managing data analyses, simulations, and other computations in almost every scientific domain [2]. It is thus not surprising that workflows have been the target of a large number of research and development activities. These activities are diverse, including the design of resource management and scheduling algorithms, the development of runtime systems to execute workflows on various hardware/software stacks, the quantitative and qualitative analysis of workflow configurations to identify commonalities and differences across scientific domains, the development of workflow benchmarks and the analysis of their results, etc. In spite of the diversity of their purposes, all these activities share a common need for access to sets of *workflow instances*. A workflow instance can be actual workflow application code and data, a set of log files obtained from a workflow execution, a formal descriptions of a workflow (task compute and data volumes, data- and control dependencies between these tasks), or any combination of these. The above need typically encompasses using both workflow instances from specific workflow applications (i.e., so as to ensure that research and development is driven by real-world data) and synthetic instances generated to be representative of these applications (i.e., so as to go beyond available real-world data and explore hypothetical and/or future scenarios).

The WfCommons project [1], [3] was established to provide data and tools that cater to research and development needs of the scientific workflow community. In this paper, we report on recent developments in WfCommons, to be released in WfCommons 1.0, that aim at enriching the data and tools that it provides. More specifically, we present:

- The development of a Nextflow "tracer" to augment the set of workflow instances provided by WfCommons; and
- The development of a Dask "translator" that makes it possible to execute WfCommons workflow instances using more workflow runtime systems; and
- An example use case that demonstrates the capabilities of WfCommons in terms of experimental evaluations.

This paper is organized as follows. Section II provides an overview of WfCommons and briefly discusses related efforts. Section III, resp. Section IV, describes new workflow tracers, resp. translators, available in WfCommons 1.0. Section V reports on a use case. Finally, Section VI briefly summarizes our recent accomplishments and outlines future work directions.

II. WfCommons Overview

Figure 1 depicts the main WfCommons components and how they relate to each other. WfCommons aims to make real-world workflow instances accessible, and this objective is shared by the Workflow Trace Archive, a recently established project highlighted in [4]. The Workflow Trace Archive comprises workflow instances generated by a preliminary version of WfCommons. WfCommons instances are constructed based on executions of real workflow applications on hardware platforms using several workflow runtime systems (arrow ① in the figure). Workflow runtime systems typically take as input some description of the workflow that is executed and generate execution logs. From these, it is possible to trace the workflow execution so as to generate re-usable workflow instances (arrow ② in the figure). Several such workflow instances have been collected in this manner based on execution of

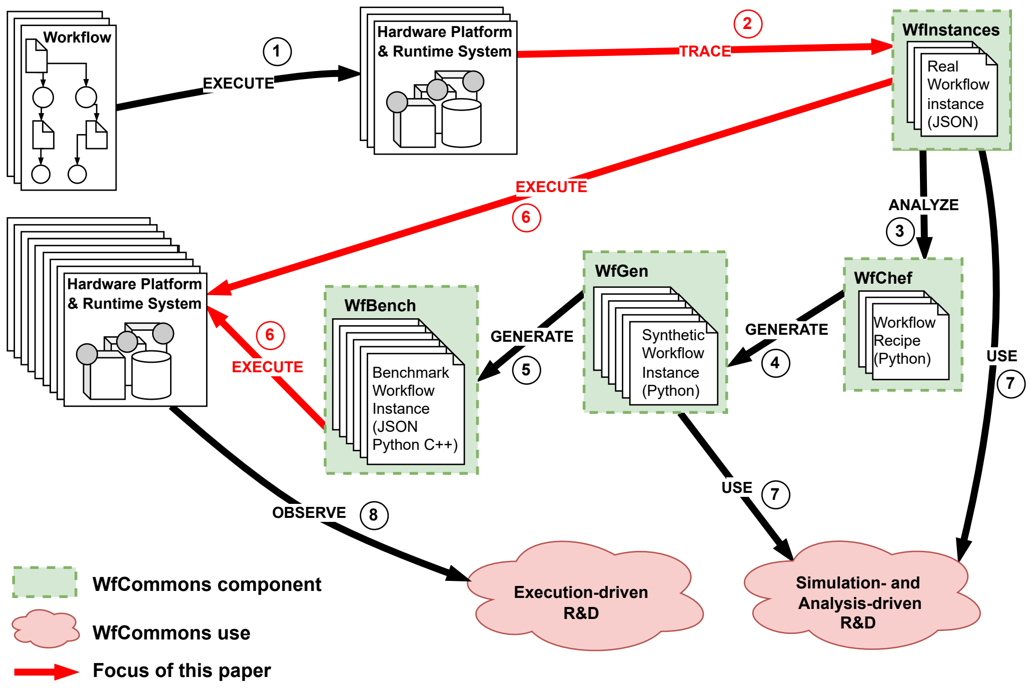
Workflow
EXECUTE
1
Hardware Platform & Runtime System
TRACE
2
WfInstances
Real Workflow instance (JSON)
ANALYZE
3
WfChef
Workflow Recipe (Python)
GENERATE
4
WfGen
Synthetic Workflow Instance (Python)
GENERATE
5
WfBench
Benchmark Workflow Instance (JSON Python C++)
EXECUTE
6
Hardware Platform & Runtime System
USE
7
OBSERVE
8
Execution-driven R&D
Simulation- and Analysis-driven R&D
WfCommons component
WfCommons use
Focus of this paper

Fig. 1. Overview of WfCommons components and their usage.

workflow configurations from 14 different applications with the Makeflow [5], Nextflow [6], and Pegasus [7] systems. In total, 170 workflow instances have been generated and are available on GitHub [8]–[10].

In WfCommons, workflow instances are described in JSON following a particular schema called WfFormat [11]. This format includes information not only about the workflow's execution on the platform on which the instance was obtained, but also about the workflow's platform-independent structure and specification, which is in contrast to the format used by the Workflow Trace Archive. Other workflow description formats have been proposed, including the popular Common Workflow Language (CWL) [12]. The WfCommons format is inspired by CWL, but encodes additional information regarding the workflow's execution and the hardware platform on which that execution took place.

Using real-world workflow instances for performing research and development is compelling but has limitations. This is because the set of available instances is necessarily limited, which in turn constraints the scope of the results obtained with these instances. For instance, a set of real-world instances for a particular scientific application may be available for only relatively small numbers of tasks, but it is often necessary to evaluate algorithms/systems with larger instances so as to assess scalability. Obtaining real-world instances at these larger sizes may not be easily doable without domain expertise. The need for generating synthetic, but representative, workflow instances has been clearly identified in the workflow community and several authors have proposed methods and tools for this purpose [13]–[19]. Improving upon these previous results, as explained in [20], WfCommons includes a component called WfChef, which takes as input sets of real-world workflow instances of different sizes obtained for a particular application, analyzes these instances, and produces so-called "workflow recipes" (arrow ③ in the figure). A workflow recipe is data and code that together describe the patterns and sub-structures found within workflow instances from a single particular application. A recipe for a particular workflow application can then be used by another WfCommons component called WfGen to generate arbitrary numbers of synthetic workflow instances of (almost) arbitrary sizes (arrow ④ in the figure). With these two components it is thus possible to generate representative synthetic workflow instances completely automatically without need for any expert application knowledge.

Real-world workflow instances archived in WfInstances or synthetic workflow instances generated by WfGen can be used directly for research and development purposes, such as driving simulations with and performing analysis of workflow instances (arrows ⑦ in the figure). But for other purposes it is necessary to execute workflow instances on hardware platforms using workflow runtime systems. For instance, this is the case for benchmarking/comparing these platforms and systems over a range of workflow scenarios, or for validating results obtained from workflow execution simulations or from analyses of workflow instances. One option is to use real-world workflow instances as benchmarks, which limits the scope of the obtained results to these particular workflow instances, and requires that all application software be installed. An alternative is to automatically generate executable workflow benchmarks. This is the approach implemented in a WfCommons component called WfBench. WfBench takes as input an arbitrary workflow instance (typically a synthetic workflow instance generated by WfGen), and automatic produces an executable workflow benchmark. The workflow tasks are replaced by a benchmark program that can be configured to mimic a range of CPU and memory usage behaviors [21]. The generated workflow benchmark can be executed without installing any software besides WfCommons. The execution of workflows instances shown by arrows ⑥ in the figure, can be performed on hardware platforms and with runtime systems that differ from those used to obtain the original instances (arrow ①). These executions can then be observed, e.g., via inspection of runtime system logs or other means for research and development purposes (arrow ⑧ in the figure).

While all components above have been available since the previous WfCommons release (v0.8), in the latest release (v1.0), we have made new developments to allow new modalities for steps shown by arrows ② and ⑥ in Figure 1, which are the topics of the next two sections.

III. Workflow Instance Tracers

A. Motivation

The most fundamental WfCommons component is WfInstances, since it provides real-world workflow instances that can be used directly or passed to WfChef to generate synthetic workflow instances. Each instance encodes both structural and execution information about the workflow. The structural information includes a list of workflow tasks, where each workflow task is described by a unique name, a path to an executable, a list of command-line arguments provided to the task, and a set of input and output files, each with a unique name and a size. The execution information includes a description of the hardware platform on which the workflow was executed and an execution time for each task. Given a workflow executed using a runtime system on some hardware platform, all the necessary information is available from the input provided to this runtime system (typically some description of the workflow's structural information) and from the output it produces (typically a set of log files). To automate the process of adding instances to WfInstances, we have implemented workflow execution "tracers" that generate workflow instances based on the runtime system input/output (arrow ② in Figure 1).

The current workflow model in WfCommons, as formalized by the WfFormat JSON schema, is a static workflow in which all tasks are known ahead of time and all data-dependencies are via input/output files. This model fits traditional workflow runtime systems, e.g., Pegasus [7] and Makeflow [5]. We have developed tracers for these two runtime systems. These tracers parse input files and log files so as to construct instances, and have been used to contribute instances to WfInstances. While for these two runtime systems developing a tracer was straightforward, this is not necessarily the case in general. Modern runtime systems use a different workflow model [22]. For instance, the input to the runtime system can be programmatic rather than purely descriptive, in which case the structure of the workflow is not known a-priori. Data dependencies may not necessarily be file-based but also based on some more general notion of data communication. Finally, runtime systems may produce logs that do not contain all the information necessary for re-constructing a workflow instance.

B. Nextflow Tracer

A popular runtime system for which the difficulties outlined in the previous section occur is Nextflow [6]. Nextflow is used heavily in the field of bioinformatics. Notably, a large set of Nextflow workflows (a.k.a. "pipelines") is available online [23]. These workflows are maintained, documented, and containerized, so that it is straightforward to execute them as is, which provides a large set of real-world workflow instances that could be contributed to WfInstances.

Nextflow uses the dataflow programming model. A Nextflow workflow is defined by a user script (e.g., bash, Python) that sets up processes that may communicate via input and output channels. These channels are used to pass data between processes. If processes need to exchange data stored in files, then a shared file system is required and messages are exchanged that contain globally visible file paths. The workflow can execute on the user's machine or in a distributed manner using various compute back-ends (e.g., SLURM, AWS Batch, Google Cloud Batch).

A preliminary Nextflow tracer was developed as part of WfCommons 0.8, and was used to produce the workflow instances available at [9]. This tracer operated by parsing the execution's standard output and the log files produced by Nextflow as is, which posed several challenges. First, it was not possible to fully determine the workflow's task dependency structure and in some cases led to task graphs with cycles. Second, although the number of bytes read/written by each task is logged, parsing the log files did not make it possible to determine which of that data was from input and output files, what these files may have been, and what file sizes were. This is in part due to Nextflow providing non-file based data-dependencies between tasks via "data channels". As a result, Nextflow instances provided as part of WfInstances with the tracer in WfCommons 0.8 are best-effort attempt and come with limitations.

In WfCommons 1.0, we implemented a Nextflow tracer that addresses the aforementioned limitations. This new tracer requires that the Nextflow source code be modified, so that each task logs the input and output messages received via its input and output data channels, respectively. The modifications are minimal, and for now they are listed in the tracer's documentation, the intent being to contribute them to Nextflow via a pull request. The tracer parses these log messages to determine the location and size of the input and output files for all tasks, based on file paths contained in exchanged data channel messages. All other information necessary to construct a WfCommons workflow instance is provided by Nextflow's logging and visualization output. Specifically, we utilize Nextflow's execution logs to obtain profile information for each task (e.g., runtime, memory requested). Also, it turns out that Nextflow produces a representation of the workflow's process graph in DOT format. This is a Directed Acyclic Graph (DAG) in which each vertex is a process, which may correspond to one or more workflow tasks (i.e., these tasks all correspond to the invocation of the same executable). This output from Nextflow is intended to produce a graphical rendering of the process graph, but we use it to reconstruct data channel dependencies between processes.

The tracer in WfCommons 1.0 improves upon that available in WfCommons 0.8, but has limitations due to the use of data channels. First, because the current version of WfFormat does not support data channels, the generated workflow instances do not include data payloads that correspond to non-file data dependencies between workflow tasks. Thus, the overall data footprint specified in the generated workflow instance is a lower bound on that of the actual Nextflow workflow. Second, the Nextflow profiling data for each task includes the number of bytes read and written by the task. Based on our investigation, it is not easy to determine whether and/or when these numbers of bytes include non-file data.

Manual inspection of the workflows we have traced to date indicates that non-file data passed via data channels is a small fraction of the overall data footprint of the workflow. Using the produced workflow instances for analysis or to drive simulation should produce realistic results. However, since information about data channel use is lost in the workflow instances when encoded in the JSON WfFormat schema, these workflow instances cannot be executed again using Nextflow or any other workflow runtime system. In other words, for these instances, top arrow ⑥ in Figure 1 is not feasible for the workflow instances produced by the Nextflow tracer.

The WfCommons Nextflow tracer is available on GitHub [24] and has been used to produce workflow instances for 15 different Nextflow workflows.

IV. Workflow Instance Translators

A. Motivation

Executing WfCommons workflow instances on real-world platforms is useful for at least two purposes. First, one may wish to execute real-world workflow instances using a different runtime system than the one that was used to obtain the instance. This is because not all workflow runtime systems are installed, or easily installable, on all platforms (for instance, some institutions prohibit installing runtime systems that require superuser privileges). Second, one may wish to conduct benchmarking campaigns by executing large numbers of workflow benchmarks (e.g., as generated by WfBench) on different platforms using different runtime systems.

Given the above, there is a strong incentive to develop WfCommons workflow instance "translator" for popular workflow runtime systems. A translator takes as input a workflow instance and produces as output all configuration files and/or code necessary to execute the instance using a target runtime system. This provides instance portability across runtime systems since a real-world workflow instance constructed from an execution with some runtime system (arrow ① in Figure 1) can be executed seamlessly with another runtime system (top arrow ⑥ in Figure 1). Furthermore, WfCommons translators can be used to execute any synthetic benchmark workflow instance produced by WfGen, without the need to install any scientific application software (bottom arrow ⑥ in Figure 1).

WfCommons 0.8 includes translators for Pegasus [7] and Swift/T [25]. Both these runtime systems employ a static Directed Acyclic Graph (DAG) model of the workflow, which directly maps to the WfFormat JSON descriptions of our workflow instances, and thus rendered the development of these translators straightforward. A popular runtime system that supports workflows is Dask [26], [27]. Dask has become a standard tool in the Python Data Science ecosystem, supports a variety of compute back-ends, and it is currently used routinely at many institutions and compute facilities. With Dask users develop workflows as Python programs, which affords more flexibility than DAG-based workflow runtime systems. The latest WfCommons release includes a Dask translator, briefly described in the next section, that makes it possible to execute any WfCommons workflow instance using Dask.

B. Dask Translator

In the Dask framework, a workflow is created by implementing each workflow task as a Python function. This function is then passed, along with its arguments, as a callback function to Dask's `submit()` method (short for `dask.distributed.Client.submit()`), which returns a `Future` (short for `dask.distributed.Future`). The `Future` points to the function's invocation and resolves whenever the function returns from that invocation. Importantly, the arguments passed to the function can include `Futures`, which makes it possible to implement control-dependencies between function invocations, and thus between workflow tasks. `Future` objects are promises of completion of a task and are managed transparently by the Dask runtime.

The WfCommons Dask translator takes as input a workflow instance in the WfFormat JSON format and produces as output a Python program that uses the Dask API and can be run to execute the workflow. Control-dependencies between tasks are realized through the use of a single callback function `execute_task()` that is passed to `submit()` and called to invoke the execution of each workflow task. The first argument passed to `execute_task()` is the function that executes the task.

The generation of the Python program that can be run to execute the workflow is performed as follows. First, information regarding how each task should be executed is collected from workflow instance. The translator sorts the workflow tasks in topological order according to task-dependencies specified in the workflow JSON document. For each task in this order, code is generated to call `submit()` using the aforementioned `execute_task()` callback function, passing it task-specific arguments and the list of `Futures` on which the task depends. These `Futures` are the return values of previous calls to `submit()` for the task's parents. Third, the translator generates code to wait on all the `Futures` returned by calls to `submit()` for the workflow's exit tasks.

TABLE I
BWA WORKFLOW BENCHMARK EXECUTION RESULTS OBTAINED WITH PEGASUS AND DASK ON 3 CHAMELEON CLOUD 48-CORE COMPUTE NODES. ALL RESULTS ARE AVERAGED OVER 5 TRIALS, WITH COEFFICIENTS OF VARIANCE IN PERCENTAGE SHOWN IN PARENTHESES.

	Pegasus			Dask		
#tasks	makespan	pre-delay	post-delay	makespan	pre-delay	post-delay
198	0h26m38s (5.6%)	40.0s (0.0%)	38.6s (6.5%)	0h22m56s (7.7%)	0.2s (2.0%)	1.5s (1.5%)
498	0h40m29s (6.3%)	40.2s (1.1%)	39.4s (4.2%)	0h32m11s (3.2%)	0.3s (4.5%)	3.7s (0.6%)
998	0h59m17s (0.7%)	40.2s (1.1%)	38.2s (4.7%)	0h53m28s (2.6%)	0.5s (2.0%)	7.4s (0.8%)
1,998	1h42m48s (3.0%)	40.4s (1.4%)	39.4s (5.8%)	1h29m15s (0.5%)	0.9s (1.6%)	14.9s (0.4%)

When the generated program is executed, all workflow tasks are concurrently submitted for execution to Dask. Initially, only the workflow's entry tasks are ready to execute because `submit()` is passed an empty list of futures. A task becomes ready only when the futures of all of its parents have resolved, which will then trigger its execution as soon as idle compute resources are available.

V. Example Use Case

To illustrate the capabilities afforded by WfCommons, in this section we describe an example use case. The objective is to evaluate two workflow runtime systems, Pegasus [7] and Dask [26], when used to execute workflows from the Bioinformatics application Burrows-Wheeler Aligner (BWA) tool [28]. BWA is a software package for mapping low-divergent sequences against a large reference genome, such as the human genome. BWA workflows have been developed based on the Makeflow [5] runtime system, as available in a GitHub repository [29]. 15 of these workflows were executed using Makeflow in December 2020 on the Chameleon Cloud testbed [30]. Based on the Makeflow input files and logs, WfCommons instances were constructed and made publicly available as WfInstances (arrows ① and ② in Figure 1).

In this example use case, we use WfChef to generate workflow recipes that describe the overall structure and substructures found in these 15 workflow instances (arrow ③ in Figure 1). Using this recipe, we use WfGen to generate 4 synthetic workflow instances (arrow ④ in Figure 1). We ask WfGen to generate instances with 200, 500, 1000, and 2000 tasks. Because WfGen generates these instances by re-using and replicating particular substructures identified by WfChef in real-world workflow instances, it cannot always generates workflow instances with the exact number of tasks specified. In this case, the generated instances have 198, 498, 998, and 1,998 tasks. Using these instances, we then used WfBench to generate corresponding workflow benchmarks. These benchmarks are generated in a way that respects the workflow structure, but replaces the tasks' executables by an invocation of a CPU-Memory benchmark program with a specified amount of work to perform, and replaces the tasks' input/output file with arbitrary files to that the total data footprint of the workflow is also specified. In our case, each workflow task runs by itself in about 35 seconds on a single core of the platform used to run the benchmark (described hereafter), and the total workflow footprint is set to 100GB.

Although the original workflows were executed using Makeflow, the generated benchmarks can be executed using any of the WfCommons translators. We use the Pegasus translator (available since WfCommons 0.8) and the recently developed Dask translator (described in Section IV-B). Using these two translators, we run the workflow benchmarks on the Chameleon Cloud testbed. Specifically, we execute the benchmarks on 3 48-core worker nodes (2.60GHz Intel Skylake), for a total of 144 cores that execute workflow tasks. Workflow executions in Pegasus or in Dask only have one technical difference: Under the hood, Pegasus uses HTCondor [31] to execute workflow tasks on its worker nodes. This requires the staging and therefore the transfer of data files and executables from and to the coordinator file system, that is the node running HTCondor `schedd`. Conversely, a shared file system (NFS in our case) was required to perform the workflow executions in Dask. In both cases, the coordinator (the HTCondor `schedd` and the Dask scheduler) was running on a dedicated node that performed no computation.

Table I shows results obtained by executing the four BWA workflow benchmarks, reporting on the workflow *makespan* (elapsed time between the submission of the workflow to the runtime system and its completion, i.e., the overall execution time as perceived by the user), the *pre-delay* (time elapsed between the submission of the workflow and the beginning of the execution of the first workflow task), and the *post-delay* (time elapsed between the completion of the last workflow task and the completion of the workflow). The metrics are computed from runtime system log files, based on back-to-back benchmark executions on dedicated compute nodes, using 5 trials for each execution. The results show that Pegasus executions have significantly higher overhead when compared to Dask executions (Pegasus makespans are between 16% and 25% longer than their Dask counterparts). The higher pre- and post-delay values with Pegasus are due to it performing staging / cleanup of data/executables on the compute nodes at the beginning / end of the execution, while Dask instead uses a shared file system. Larger makespans with Pegasus may also be due to HTCondor, which necessarily adds overhead.

The results in Table I in no way provide an in-depth comparison of Pegasus and Dask. Conducting more experiments and analyzing execution logs in detail would be necessary for a full-fledged comparison. Our objective here is to illustrate that, using WfCommons, performing this kind of experimental study is low-labor. Specifically, generating a workflow benchmarks is done with 2 lines of Python. Executing each benchmark with the Pegasus translator is done with 4 lines of Python. Executing each benchmark with the Dask translator is done with 2 Shell commands. Overall, the entire code a WfCommons user would have to write to generate the results in Table I consists of less than 40 lines of Python/Shell.

VI. Conclusion

In this paper, we have provided an overview of WfCommons and of its components, have described recent developments that aim to augment the set of workflow instances that WfCommons can provide and to make these instances executable with modern workflow runtime systems, and described an illustrative use case. WfCommons was initially designed and developed with DAG-based and file-based workflow runtime systems in mind. Hence, WfFormat was not designed to support dynamic/programmatic workflows or workflows in which task data-dependencies are not always file-based. As explained in Section III, this design hindered the implementation of the Nextflow tracer, as Nextflow supports non-file data-dependencies. Furthermore, although the Dask translator described in Section IV uses Dask for executing static workflows, Dask provides the ability to execute workflows in which tasks are created dynamically at runtime rather than at compile time, and Dask does not impose that task dependencies be file-based. Consequently, a future direction is to augment WfFormat to support a broader range of modern workflow models [22], which will require reviewing workflow descriptions modalities in popular/emerging workflow runtime systems as well as recognized standards such as CWL [12].

Acknowledgment

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This research was partially supported by NSF Award #2106059. Finally, we thank the NSF Chameleon Cloud for providing time grants to access their resources.

References

[1] "WfCommons Project," https://wfcommons.org, 2023.

[2] C. S. Liew, M. P. Atkinson, M. Galea, T. F. Ang, P. Martin, and J. I. V. Hemert, "Scientific workflows: moving across paradigms," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, pp. 1–39, 2016.

[3] T. Coleman, H. Casanova, L. Pottier, M. Kaushik, E. Deelman, and R. Ferreira da Silva, "WfCommons: A Framework for Enabling Scientific Workflow Research and Development," *Future Generation Computer Systems*, vol. 128, pp. 16–27, 2022.

[4] L. Versluis, R. Mathá, S. Talluri, T. Hegeman, R. Prodan, E. Deelman, and A. Iosup, "The Workflow Trace Archive: Open-Access Data From Public and Private Computing Infrastructures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 9, pp. 2170–2184, 2020.

[5] M. Albrecht, P. Donnelly, P. Bui, and D. Thain, "Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids," in *1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*. ACM, 2012, p. 1.

[6] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, "Nextflow enables reproducible computational workflows," *Nature Biotechnology*, pp. 316–319, 2017.

[7] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger, "Pegasus: a Workflow Management System for Science Automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015.

[8] "WfCommons Makeflow Execution Instances," https://github.com/wfcommons/makeflow-instances, 2023.

[9] "WfCommons Nextflow Execution Instances," https://github.com/wfcommons/nextflow-instances, 2023.

[10] "WfCommons Pegasus Execution Instances," https://github.com/wfcommons/pegasus-instances, 2023.

[11] "WfFormat: The WfCommons JSON Schema," https://github.com/wfcommons/wfformat, 2023.

[12] M. R. Crusoe, S. Abeln, A. Iosup, P. Amstutz, J. Chilton, N. Tijanić, H. Ménager, S. Soiland-Reyes, B. Gavrilović, C. Goble *et al.*, "Methods included: Standardizing computational reuse and portability with the common workflow language," *Communications of the ACM*, vol. 65, no. 6, pp. 54–63, 2022.

[13] "DAGGEN: a synthetic task graph generator," https://github.com/frs69wq/daggen, 2021.

[14] M. A. Amer and R. Lucas, "Evaluating Workflow Tools with SDAG," in *SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 2012, pp. 54–63.

[15] D. G. Amalarethinam and G. J. Mary, "Dagen-a tool to generate arbitrary directed acyclic graphs used for multiprocessor scheduling," *International Journal of Research and Reviews in Computer Science*, vol. 2, no. 3, p. 782, 2011.

[16] D. G. Amalarethinam and P. Muthulakshmi, "Dagitizer–a tool to generate directed acyclic graph through randomizer to model scheduling in grid computing," in *Advances in Computer Science, Engineering & Applications*. Springer, 2012, pp. 969–978.

[17] W. M. van Der Aalst, A. H. Ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distributed and parallel databases*, vol. 14, no. 1, pp. 5–51, 2003.

[18] D. Garijo, P. Alper, K. Belhajjame, O. Corcho, Y. Gil, and C. Goble, "Common motifs in scientific workflows: An empirical analysis," *Future Generation Computer Systems*, vol. 36, pp. 338–351, 2014.

[19] R. Ferreira da Silva, W. Chen, G. Juve, K. Vahi, and E. Deelman, "Community resources for enabling and evaluating research in distributed scientific workflows," in *10th IEEE International Conference on e-Science*, ser. eScience'14, 2014, pp. 177–184.

[20] T. Coleman, H. Casanova, and R. Ferreira da Silva, "Wfchef: Automated generation of accurate scientific workflow generators," in *17th IEEE eScience Conference*, 2021, pp. 159–168.

[21] T. Coleman, H. Cansanova, K. Maheshwari, L. Pottier, S. R. Wilkinson, J. Wozniak, F. Suter, M. Shankar, and R. Ferreira da Silva, "WfBench: Automated Generation of Scientific Workflow Benchmarks," in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2022, pp. 100–111.

[22] R. Ferreira da Silva, R. M. Badia, V. Bala, D. Bard, T. Bremer *et al.*, "Workflows Community Summit 2022: A Roadmap Revolution," Oak Ridge National Laboratory, Tech. Rep. ORNL/TM-2023/2885, 2023.

[23] "nf-core Nextflow pipelines," https://nf-co.re/pipelines, 2023.

[24] "WfCommons Nextflow workflow tracer," https://github.com/wfcommons/nextflow_workflow_tracer, 2023.

[25] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/T: Large-scale application composition via distributed-memory dataflow processing," in *Proc. IEEE/ACM International Symp. on Cluster, Cloud, and Grid Computing*. IEEE, 2013, pp. 95–102.

[26] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in science conference*, no. 130-136, 2015.

[27] "The Dask project," https://www.dask.org/, 2023.

[28] "Burrows-Wheeler Aligner," https://bio-bwa.sourceforge.net/, 2023.

[29] "BWA Makeflow workflows," https://github.com/cooperative-computing-lab/makeflow-examples/tree/master/bwa, 2023.

[30] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons Learned from the Chameleon Testbed," in *Proc. of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[31] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the Condor experience," *Concurrency - Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.