Automated Generation of Scientific Workflow Generators with WfChef

Tainã Coleman[a], Henri Casanova[b], Rafael Ferreira da Silva[a,c,*]

[a] *University of Southern California, Department of Computer Science, Los Angeles, CA, USA*
[b] *Information and Computer Sciences, University of Hawaii, Honolulu, HI, USA*
[c] *National Center for Computational Sciences, Oak Ridge National Laboratory, Oak Ridge, TN, USA*

Abstract

Scientific workflow applications have gained significant importance, and their automated and efficient execution on large-scale computing platforms has been the subject of extensive research and development. For these efforts to be successful, a solid experimental methodology is needed to evaluate workflow algorithms and systems. A foundation for this methodology is the availability of realistic workflow instances. Although public repositories provide workflow instances for a few scientific applications, these are limited in scope, and workflow instances are not available for all application scales of interest. To address this limitation, previous work has developed generators of synthetic workflow instances of arbitrary scales. Despite being popular, the implementation of these generators is a manual and labor-intensive process that requires expert application knowledge. As a result, these generators only target a handful of applications, even though there are hundreds of workflow applications in production.

We introduce *WfChef*, a fully automated framework for constructing a synthetic workflow generator for any scientific application. Based on an input set of workflow instances for a particular application, WfChef automatically produces a synthetic workflow generator. To measure the realism of the generated workflows, we define and evaluate several metrics. Using these metrics, we compare the realism of the workflows generated by WfChef generators to that of the workflows generated by the previously available, hand-crafted generators. We find that WfChef generators not only require zero development effort (because they are automatically produced), but also generate workflows that are more realistic than those generated by hand-crafted generators.

Keywords: Scientific Workflows, Synthetic Workflow Generation, Workflow Management Systems, Workflow Pattern Detection, Automatic Workflow Generation, Realistic Synthetic Workflow Instance

1. Introduction

In the past several decades, scientific workflows have supported some of the most significant discoveries [1] in an abundance of scientific domains. Many computationally intensive scientific applications have been framed as *scientific workflows* that execute on various compute platforms and at various platform scales [2]. Scientific workflows are typically described as Directed Acyclic Graphs (DAGs) in which vertices represent tasks and edges represent dependencies between tasks, as defined by application-specific semantics. As workflows continue to be adopted by scientific projects and user communities, they are becoming more complex. Today's production workflows can be composed of millions of individual tasks that execute for milliseconds to hours, and that can be single-threaded programs, multi-threaded programs, tightly coupled parallel programs (e.g., MPI programs), or loosely coupled parallel programs (e.g., MapReduce jobs), all within a single workflow [3]. The automated execution of workflows on these platforms have been the object of extensive research and development, as seen in the number of proposed workflow resource management and scheduling approaches[1], and the number of developed workflow systems (a self-titled "incomplete" list [4] points to 300+ distinct systems, although many of them are no longer in use). Thus, in spite of workflows and workflow systems being used in production daily, workflow computing is an extremely active research and development area, with many remaining challenges [3, 5, 6, 7].

Addressing these challenges requires a solid experimental methodology for evaluating and benchmarking workflow algorithms, platforms, and runtime systems [8]. This methodology requires sets of representative workflow instances. One approach is to extract workflow structures from real-world execution logs, as we have done in previous work [9, 10]. This has resulted in a repository that hosts ~20 workflow instances for each of several scientific applications [11]. These instances can be and have been used by researchers for analyzing the structure of scientific workflow applications, for identifying commonalities across these applications, for providing workflow instances as input to simulators of workflow executions developed using

[*] Corresponding address: Oak Ridge National Laboratory, P.O. Box 2008, Oak Ridge, TN, USA 37831

[**] This manuscript has been authored in part by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The publisher acknowledges the US government license to provide public access under the DOE Public Access Plan (`http://energy.gov/downloads/doe-public-access-plan`).

Email addresses: `tgcolema@usc.edu` (Tainã Coleman), `henric@hawaii.edu` (Henri Casanova), `silvarf@ornl.gov` (Rafael Ferreira da Silva)

[1] The IEEE Xplore digital database includes 117 and 92 articles with both the words "Workflow" and "Scheduling" in their title for 2021 and 2022, respectively.

various simulation frameworks, which in turn makes it possible to evaluate Workflow Management Systems (WMSs) designs and the scheduling and resource management algorithms implemented in these systems using realistic workflow configurations.

Real workflow instances are, by definition, representative of real applications. However, they are limited in number and scope, and creating new real instances requires significant time, resources, and expertise. To overcome this limitation, in previous work we have developed tools for generating synthetic workflows by extrapolating the patterns seen in real workflow instances. The work in [9] presents a synthetic workflow generator for four workflow applications, which has been used extensively by researchers[2]. The method for generating the synthetic workflows was ad-hoc and based on expert knowledge and manual inspection of real workflow instances. Our more recent generator in [10] improves on the previous generator by using information derived from statistical analysis of execution logs. It was shown to generate more realistic workflows than the earlier generator, and in particular to preserve key workflow features when generating workflows at different scales [10]. The main drawback of these two generators is that implementing the workflow generation procedure is labor-intensive. Generators are manually crafted for each application, which not only requires significant development effort (several hundreds of lines of code) but also, and more importantly, expert knowledge about the scientific application semantics that define workflow structures. As a result, this approach is not scalable if synthetic workflow instances are to be generated for a large number of scientific applications.

We introduce ***WfChef***, a framework that automates the process of constructing a synthetic workflow generator for any given workflow application. WfChef takes a set of real workflow instances as input and outputs the code of a synthetic workflow generator for that application. By analyzing the task graphs of the real workflows, WfChef identifies subgraphs that represent fundamental task dependency patterns. It then uses this information, along with task type frequencies, to generate a synthetic workflow generator that can create realistic synthetic workflow instances with an arbitrary numbers of tasks. In this work, we evaluate the realism of the synthetic workflows generated by WfChef, both in terms of workflow structure and execution behavior. Note that WfChef is a core module of a larger framework called WfCommons [12], but it also has standalone value. For example, in [13], WfChef was used standalone to generate synthetic workflow instances to augment the training set for GCNScheduler, a novel machine learning approach to workflow scheduling on heterogeneous networks. Specifically, this work makes the following contributions[3]:

1. We describe the overall architecture of WfChef and the algorithms it uses to analyze real workflow instances and produce a workflow generator;
2. We quantify the realism of the generated workflows when compared to real workflow instances, in terms of abstract graph similarity metrics and of the realism of simulated workflow executions;
3. We compare the realism of the generated workflows to that of the workflows generated by the original workflow generator in [9] and by the more recent generator in [10];
4. We show that for complex scientific workflows using only a fraction of the real workflow instances available still yields realistic synthetic instances, in addition to being significantly faster;
5. Our key finding is that the generators automatically produced by WfChef lead to equivalent or improved (often vastly) results when compared to the previously available, manually implemented, workflow generators.
6. We implemented WfChef as part of an open source framework [12] that provides foundational tools for scientific workflow research and development, and that has already supported 10+ research articles.

2. Related Work

Scientific workflow configurations, both inferred from real-world executions and synthetically generated, have been used extensively in the workflow research and development community, in particular for evaluating resource management and scheduling approaches. As scientific workflows are typically represented as Directed Acyclic Graphs (DAGs), several tools have been developed to generate random DAGs, based on specified ranges of values for various parameters [15, 16, 17, 18]. For instance, DAGGEN [15] and SDAG [16] generate random DAGs based on parameters such as the number of tasks, the width, the edge density, the maximum number of levels that can be spanned by an edge, the data-to-computation ratio, etc. Similarly, DAGEN [17] generates random DAGs, but does so for parallel programs in which the task computation and communication payloads are modeled according to classic parallel programs. DAGITIZER [18] is an extension of DAGEN for grid workflows where all parameters are randomly generated. Although these generators can produce a very diverse set of DAGs, these DAGs may not resemble those found in actual scientific workflows as they do not capture patterns defined by application-specific semantics.

An alternative to random generation is to generate DAGs based on the structure of real workflows for particular scientific applications. In [19], over forty workflow patterns are identified for addressing business process requirements (e.g., sequence, parallelism, choice, synchronization, etc.). Although several of these patterns can be mapped to some extent to structures that occur in scientific workflows [20], they do not fully capture these structures. In particular, they do not necessarily respect the ratios of different types of particular workflow tasks in these structures. This is important because a workflow structure is not only defined by a set of vertices and edges, but also by the task type (e.g., an executable name) of each vertex. The work in [21] focuses on identifying workflow "motifs" based on observing the data created and used by workflow tasks so as

[2] To date, 350+ bibliographical references to the research article and/or the software repository's URL.

[3] A preliminary shorter version of this paper appears in the proceedings of the 2021 IEEE eScience Conference [14].

to reverse-engineer workflow structures. These motifs capture workflow (sub-)structures, and can thus be used for automated workflow generation. Unfortunately, identifying these motifs is an arduous manual process [21]. In our previous work [9], we developed a tool for generating synthetic workflow configurations based on real-world workflow instances. Although the overall structure of generated workflows was reasonably realistic, we found that workflow execution (simulated) behavior was not (see [10] and also results in Section 5.3). In [10], we developed an enhanced version of that earlier generator in which task computational loads are more accurately captured by using statistical methods. As a result, the generated synthetic workflows are more realistic. While the task computational load characterization is automated, the DAG-generation procedure is labor-intensive because generators are manually crafted and rely on expert knowledge of the workflow application. Identifying workflow patterns/structures to generate synthetic workflow instances has several applications, a crucial one being benchmarking. Task Bench [22] is a parameterized benchmark designed to assess the performance of distributed programming systems. Task Bench offers a number of sample task graphs (workflows) that represents some of the most common patterns found in workflow applications. But real workflow applications, although they do include some of these patterns, typically combine them in complex structures, and capturing these structures to generate realistic task graphs is challenging.

All above works attempt to generate synthetic workflow instances that are representative of real-world workflows, so as to enable research and development activities, including benchmarking. The goal of this work is to automate synthetic workflow instance generation. Automatic (non-synthetic) workflow instance generation has been proposed to help users translate their conceptual workflows into concrete workflows that can be executed with some target runtime system, or Workflow Management System (WMS). This is movitaged by the fact that there is a large number of such systems, that users may want to use multiple such systems, and that doing so is labor-intensive. The work in [23] proposes a tool, wfGenes, by which a user constructs an abstract workflow description. This abstract description is then translated into a workflow description in the format of some target WMS, and then validated against the workflow description schema provided by that WMS. This approach has been used for real-world HPC use cases to generate workflows that are executable with two distinct production WMSs. The work in [23] is orthogonal but complementary to this work. While wfGenes does not generate synthetic workflow instances, the synthetic workflow instances produced by the approach proposed in this work could be provided as input to wfGenes so as to produce workflows that can be executed with a range of WMSs.

To the best of our knowledge, this is the first work that attempts a completely automated synthetic workflow generation approach (automated analysis of real workflow instances to drive the automated generation of synthetic workflows). Our approach makes it straightforward to generate synthetic workflows at arbitrary scales that are representative of real workflow instances for any workflow application. These synthetic workflows are key for supporting the development and evaluation of workflow algorithms. Also, they can provide a fundamental building block for the automatic generation of workflow application skeletons [24], which can then be used to benchmark workflow systems [25].

3. Problem Statement

In this paper, we focus on the automated generation of generators that aim to capture the overall structure of workflows, including task dependencies and the task graph. The primary goal of these generators is to facilitate the evaluation of scientific workflow systems and optimization techniques without the need to deploy complex applications and their software dependencies. Generating synthetic workflows that resemble real-world workflows makes it possible to broaden the scale and the scope of research and development activities that target workflow applications, platforms, systems, and algorithms.

Consider a scientific application for which a set of real workflow instances, W, is available. Each workflow w in W is a DAG, where the vertices represent workflow tasks and the edges represent task dependencies. In this work, we only consider workflows that comprise tasks that execute on a single compute node – i.e., tasks are not parallel jobs (which is the case for a large number of scientific workflow applications [3, 2, 26, 27]). More formally, $w = (V, E)$, where V is a set of vertices and E is a set of directed edges. We use the notation $|w|$ to denote the number of vertices in w (i.e., $|w| = |V|$). We assume that each workflow has a single entry vertex and a single exit vertex (for workflows that do not we simply add dummy entry/exit vertices with necessary edges to all actual entry/exit vertices). Finally, a *type* is associated to each vertex v, denoted as $type(v)$. This type denotes the particular computation that the corresponding workflow task must perform. In this work, we consider workflows in which every task corresponds to an invocation of a particular method or executable, and we simply define a vertex's type as the name of that method or executable. Several tasks in the same workflow can thus have the same type.

Problem Statement – Given W, the objective is to produce the code for a workflow generator that generates realistic synthetic workflow instances. This workflow generator takes as input an integer, $n \geq \min_{w \in W}(|w|)$. It outputs a workflow w' with $n' \geq n$ vertices that is as realistic as possible. n' may not be equal to n because real workflows for most scientific applications cannot be feasibly instantiated for arbitrary numbers of tasks. Our approach guarantees that n' is the smallest feasible number of tasks that is greater than n.

We use several metrics to quantify the realism of the generated workflow. Consider a workflow generated with the workflow generator, w', and a real workflow instance with the same number of vertices, w. The realism of workflow w' can be quantified based on DAG similarity metrics that perform vertex-to-vertex and edge-to-edge comparisons (see Section 5.2). The realism can also be quantified based on similarity metrics computed between the logs of (simulated) executions of workflows w and w' on a given compute platform (see Section 5.3).

4. The WfChef Approach

In this section, we describe our approach, WfChef. In Section 4.1, we define particular sub-DAGs in a set of workflow instances. Algorithms to detect these sub-DAGs and to use them for synthetic workflow generation are described in Section 4.2. Finally, in Section 4.3 we briefly describe our implementation of WfChef.

4.1. Pattern Occurrences

The basis for our approach is the identification of particular sub-DAGS in workflow instances for a particular application. Let us first define the concept of a *type hash*:

Definition 1 (Type hash). *Given a workflow vertex v, we define its* top-down hash*, $TD(v)$, recursively as the following string. Consider the lexicographically sorted list of the unique top-down hashes of v's successors. $TD(v)$ is the concatenation of these top-down hashes and of type(v). We define v's* bottom-up hash*, $BU(v)$, similarly, but considering predecessors instead of successors. Finally, we define v's* type hash*, $TH(v)$, as the concatenation of $TD(v)$ and $BU(v)$.*

Figure 1 is an example for a simple 9-task workflow, where TD, TU, and TH strings are shown for each vertex. The type hash of each vertex in a workflow encodes information regarding the vertex's role in the structures and sub-structures of the workflow. From now on, we assume that each vertex is annotated with its type hash. Given a workflow w, we define the type hash of w, denoted as $TH(w)$, as the set of unique type hashes of w's vertices. $TH(w)$ can be computed in $O(|w|^2 \log(|w|))$. We must calculate TD and BU hashes for all vertices because they can have the same TD and different BU, and vice-versa.

The basis of our approach is the observation that, given a workflow, sub-DAGs of it that have the same type hash are representative of the same application-specific pattern (i.e., groups of vertices of certain types with certain dependency structures, but not necessarily the same size). We formalize the concept of a *pattern occurrence* as follows:

Definition 2 (Pattern Occurrence (PO)). *Given a set of workflow instances for an application W, a* pattern occurrence *is a DAG po such that:*

- *po is a sub-DAG of at least one workflow in W;*
- *There exists at least one workflow in W with two sub-DAGs g' and g'' such that:*
  - *g' and g'' are disjoint;*
  - *$TH(g') = TH(g'') = TH(po)$;*
  - *Any two entry, resp. exit, vertices in g' and g'' that have the same type hash have the exact same parents, resp. children.*

Figure 2 shows an example workflow, where vertex types are once again indicated by colors. Based on the above definitions, this workflow contains 6 POs, each shown within a rectangular box. The two POs in the red boxes have the same type hash, and we say that they correspond to the same pattern. But note that although they correspond to the same pattern, they do not have the same number of vertices. POs can occur within POs, as is the case for the POs in the green boxes in this example. Note that a sub-DAG of the rightmost POs (the three-task PO in the red box) has the same type hash as the POs in the green boxes. In fact, it is identical to those POs (i.e., a blue vertex followed by a green vertex). But this subgraph is not a PO because it does not have a common ancestor with any of the other POs with similar type hashes.

4.2. Algorithms

WfChef consists of two main algorithms, WfChefRecipe and WfChefGenerate. The former is invoked only once and takes as input a set of workflow instances for a particular application, W, and outputs a "recipe", i.e., a data structure that encodes relevant information extracted from the workflow instances. The latter is invoked each time a synthetic workflow instance needs to be generated. It takes as input a recipe and a desired number of vertices (as well as a seeded pseudo-random number generator), and outputs a synthetic workflow instance. Both these algorithms have polynomial complexity and implement several heuristics, as described hereafter.

Algorithm 1 Algorithm to compute a recipe based on a set of real workflow instances.

```
 1: function WfChefRecipe($W$)
 2:     $POs \leftarrow \{\}$                                  ▷ dictionary of POs
 3:     for each $w \in W$ do
 4:         $POs[w] \leftarrow []$                             ▷ list of POs in $w$
 5:         for each unvisited vertex $v$ in $w$ do
 6:             mark $v$ as visited
 7:             $v'$ = an unvisited vertex s.t. $TH(v') = TH(v)$
 8:             if $v'$ is not found then continue
 9:             mark $v'$ as visited
10:             A = ClosestCommonAncestors($v$,$v'$)
11:             D = ClosestCommonDescendants($v$,$v'$)
12:             if $A = \emptyset$ or $B = \emptyset$ continue
13:             $POs[w]$.append(SubDAG($v$, $A$, $B$))
14:             $POs[w]$.append(SubDAG($v'$, $A$, $B$))
15:         end for
16:     end for
17:     $Errors \leftarrow \{\}$                               ▷ dictionary of errors
18:     for each $w \in W$ do
19:         for each $b \in W$ s.t. $|b| < |w|$ do
20:             $g \leftarrow$ ReplicatePOs ($|w|$, $b$, $POs[b]$, $POs[w]$)
21:             $Errors[b][w] \leftarrow$ Error($w$,$g$)
22:         end for
23:     end for
24:     return new $Recipe(W, POs, Errors)$
25: end function
```

WfChefRecipe pseudo-code is shown in Algorithm 1. Lines 2 to 16 are devoted to detecting all POs in W. For each w in W, the algorithm visits w's vertices (Lines 5-15). An arbitrary unvisited vertex v is visited, and another arbitrary unvisited vertex v' is found, if it exists, that has the same type-hash as v (Lines 6-7). If no such v' exists then the algorithm visits another vertex v (Line 8). Otherwise, it marks v' as visited (Line 9) and computes the set of closest common ancestor and successor vertices

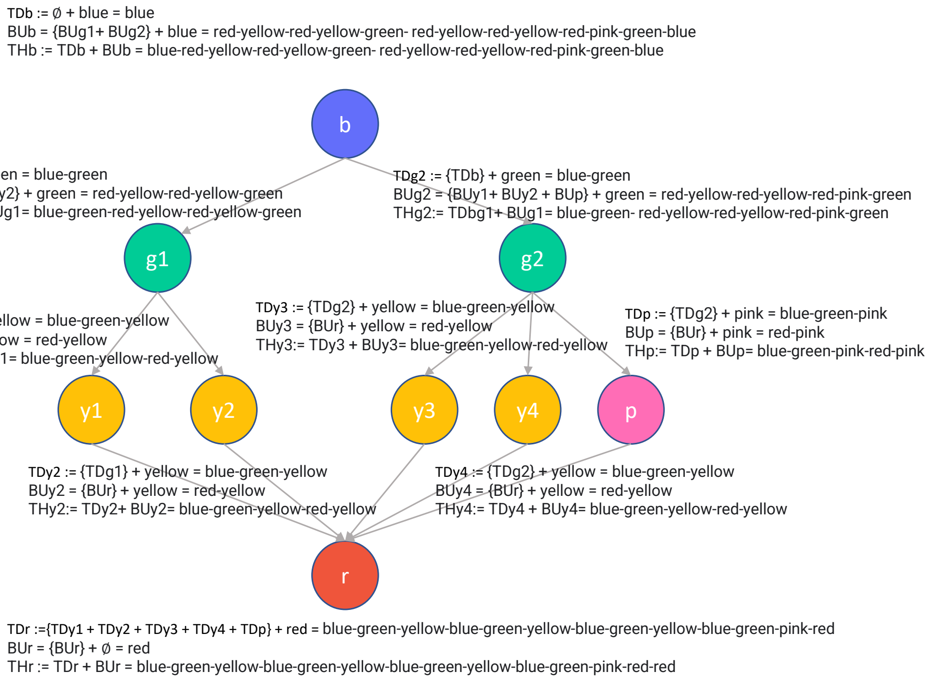

TDb := ∅ + blue = blue
BUb = {BUg1+ BUg2} + blue = red-yellow-red-yellow-green- red-yellow-red-yellow-red-pink-green-blue
THb := TDb + BUb = blue-red-yellow-red-yellow-green- red-yellow-red-yellow-red-pink-green-blue
b
TDg1 := {TDb} + green = blue-green
BUg1 = {BUy1+ BUy2} + green = red-yellow-red-yellow-green
THg1:= TDbg1+ BUg1= blue-green-red-yellow-red-yellow-green
TDg2 := {TDb} + green = blue-green
BUg2 = {BUy1+ BUy2 + BUp} + green = red-yellow-red-yellow-red-pink-green
THg2:= TDbg1+ BUg1= blue-green- red-yellow-red-yellow-red-pink-green
g1
g2
TDy1 := {TDg1} + yellow = blue-green-yellow
BUy1 = {BUr} + yellow = red-yellow
THy1:= TDy1+ BUy1= blue-green-yellow-red-yellow
TDy3 := {TDg2} + yellow = blue-green-yellow
BUy3 = {BUr} + yellow = red-yellow
THy3:= TDy3 + BUy3= blue-green-yellow-red-yellow
TDp := {TDg2} + pink = blue-green-pink
BUp = {BUr} + pink = red-pink
THp:= TDp + BUp= blue-green-pink-red-pink
y1
y2
y3
y4
p
TDy2 := {TDg1} + yellow = blue-green-yellow
BUy2 = {BUr} + yellow = red-yellow
THy2:= TDy2+ BUy2= blue-green-yellow-red-yellow
TDy4 := {TDg2} + yellow = blue-green-yellow
BUy4 = {BUr} + yellow = red-yellow
THy4:= TDy4 + BUy4= blue-green-yellow-red-yellow
r
TDr :={TDy1 + TDy2 + TDy3 + TDy4 + TDp} + red = blue-green-yellow-blue-green-yellow-blue-green-yellow-blue-green-pink-red
BUr = {BUr} + ∅ = red
THr := TDr + BUr = blue-green-yellow-blue-green-yellow-blue-green-yellow-blue-green-pink-red-red

Figure 1: Example workflow with TD, BU, and TH strings shown for each vertex. Vertex types are simply their colors (“red”, “green”, “blue”, “yellow”, “pink”) and their identifier is the color plus a number (“green1”, “green2”,“yellow1”, “yellow2”, “yellow3”, “yellow4”). ∅ denotes the empty string, and + denotes the string concatenation operator.

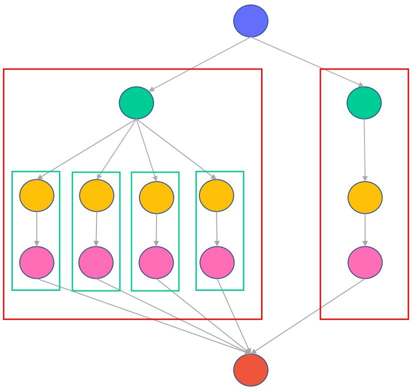


Figure 2: Example workflow with 6 POs, shown in rectangular boxes. Boxes with the same color indicate POs with identical type hashes.

for v and v' (Lines 10-11). The pseudo-code of the ClosestCommonAncestors and ClosestCommonDescendants functions is not shown as they are simple DAG traversals. If v and v' do not have at least one common ancestor and one common descendant, then the algorithm visits another vertex v (Line 12). Otherwise, two POs have been found, which are constructed and appended to the list of POs that occur in w at Lines 13 and 14. The pseudo-code for function SubDAG is not shown. It takes as input a vertex in a DAG, a set of ancestors of that vertex, and a set of descendants of that vertex. It returns a DAG that contains all paths from all ancestors to all descendants to traverse v, but from which the ancestors and descendants have been removed (along with their outgoing and incoming edges).

Lines 17 to 23 are devoted to computing a set of “errors” resulting from using a particular (smaller) base workflow to generate a larger (synthetic) workflow. The WfChef approach consists in replicating POs in a base workflow to scale up its number of vertices while retaining a realistic structure. Therefore, when needing to generate a synthetic workflow at a particular scale, it is necessary to choose a base workflow as a starting point. To provide some basis for this choice, for each $w \in W$, the algorithm generates a synthetic workflow with $|w|$ vertices using as a base each workflow in W with fewer vertices than w (Lines 12-22). The ReplicatePOs function replicates POs in a base workflow to generate a larger synthetic workflow (it is described at the end of this section). The error, that is the discrepancy between the generated workflow and w, is quantified via some error metric (the Error function) and recorded at Line 21 (in our implementation we use the THF metric described in Section 5.2). The way in which these recorded errors are used in our approach is explained in the description of WfChefGenerate hereafter. Finally, at Line 24, the algorithm returns a recipe, i.e., a data structure that contains the workflow instances (W), the discovered pattern occurrences (POs), and the above errors ($Errors$).

The pseudo-code for WfChefGenerate is shown in Algorithm 2. It takes as input a recipe (rcp) and a desired number of vertices n. At Line 2, the algorithm determines the workflow in W that has the numbers of vertices closest to n. This workflow is called $closest$. At Line 3, the algorithm finds the workflow in W that, when used as a base for generating a synthetic workflow with $|closest|$ vertices, leads to the lowest error. The intent here is to pick the best base workflow for generating a synthetic workflow with n vertices. No workflow in W may have

Algorithm 2 Algorithm for generating a synthetic workflow with n vertices based on a recipe.

```
1: function WfChefGenerate($rcp$, $n$)
2:     $closest \leftarrow w$ in $rcp.W$ s.t. $||w| - n|$ is minimum
3:     $base \leftarrow w$ in $rcp.W$ t. $rcp.Errors[w, closest]$
   is minimum
4:     $g \leftarrow$ ReplicatePOs ($n$, $base$, $rcp.POs[base]$,
           $r.POs[closest]$)
5:     return $g$
6: end function
```

exactly n vertices. As a heuristic, we choose the best base workflow for generating a synthetic workflow with $|closest|$ vertices, based on the errors computed at Lines 17 to 23 in Algorithm 1. The synthetic workflow is generated by calling function ReplicatePOs at Line 4, and returned at Line 5.

Algorithm 3 Algorithm for replicating POs in a base workflow.

```
 1: function ReplicatePOs($n$, $base$, $bPOs$, $cPOs$)
 2:     $g \leftarrow base$
 3:     $prob \leftarrow \{\}$                    ▷ dictionary of probabilities
 4:     for each $po \in bPOs$ do
 5:         $nc = |\{p \in cPOs \mid TH(p) = TH(po)\}|$
 6:         $tc = |\{p \in cPOs\}|$
 7:         $nb = |\{p \in bPOs \mid TH(p) = TH(po)\}|$
 8:         $prob[po] \leftarrow (nc/tc)/nb$
 9:     end for
10:     while $|g| < n$ do
11:         $po \leftarrow$ sample from $bPO$ with distribution $prob$
12:         $g \leftarrow$ AddPO($g$, $po$)
13:     end while
14:     return $g$
15: end function
```

The pseudo-code for ReplicatePOs is shown in Algorithm 3. It takes as input a desired number of vertices (n), a base workflow ($base$), the list of POs in the base workflow ($bPOs$), and the list of POs in the workflow whose number of vertices is the closest to n ($cPOs$). The intent is to replicate POs in the base workflow, picking which pattern to replicate based on the frequency of POs for that pattern in the closest workflow. At Line 2, the algorithm first sets the generated workflow to be the base workflow. Lines 4-9 are devoted to computing a probability distribution. More specifically, for each PO in $bPOs$, the algorithm computes the probability with which this PO should be replicated. Given a PO in $bPOs$, nc is the number of POs for that same pattern in $cPOs$ (Line 5) and tc is the total number of POs in $cPOs$. Thus, nc/tc is the probability that a PO in $cPOs$ is for that same pattern. nb is the number of POs in $bPOs$ for that same pattern (Line 7). The probability of picking one of these POs in bPO for replication is thus computed as $((nc/tc)/nb)$ (Line 8). Note that this probability could be zero since nc could be zero. The algorithm then iteratively adds one PO from the base graph to the generated graph (while loop at Lines 10 to 13). At each iteration, a PO po in bPO is picked randomly with probability $prob[po]$ (Line 11), and this pattern is added to g (Line 12). The function AddPO operates as follows. Given a workflow, g, and a to-be-added PO, po, for a specific pattern, it: (i) randomly picks in g one existing PO for that same pattern, po'; (ii) adds po to the workflow, connecting its entry, resp. exit, vertices to the parent, resp. children, vertices of the corresponding entry, resp. exit, vertices of po'.

The pseudo-code in this section is designed for clarity. Our actual implementation, described in the next section, is more efficient and avoids all unnecessary re-computations (e.g., the probabilities computed in WfChefGenerate).

4.3. Implementation

We have implemented WfChef as part of a scientific workflow framework, WfCommons [12], which provides a collection of tools for analyzing workflow executions, for producing generators of synthetic workflows, and for simulating workflow executions. WfChef has been distributed as a core Python module of WfCommons[4] since version v0.6 and has already supported 10+ research articles. Specifically, the `wfchef` module defines a `Recipe` class. The constructor for that class takes as input a list of workflow instances and implements algorithm WfChefRecipe. The workflow instances are provided as files in the WfCommons JSON format[5]. The class has a public method `duplicate` that implements the WfChefGenerate algorithm, and a private method `duplicate_nodes` that implements the ReplicatePOs algorithm. WfCommons distribution packages provide a set of pre-computed recipes generated with WfChef, which are produced based on the set of workflow instances collected, curated, and distributed as part of the WfCommons framework. Users also have the option to generate their own recipes using WfChef's methods and their own set of workflow instances.

5. Experimental Evaluation

In this section, we evaluate our approach and compare it to previously proposed approaches. In Section 5.1, we describe our experimental methodology. We evaluate the realism of generated workflows based on their structure, in Section 5.2, and based on their simulated execution, in Section 5.3.

5.1. Methodology

We compare the realism of the synthetic instances generated by WfChef generators to that of instances generated with the original workflow generator in [9], which we call **WorkflowGenerator**, and with the more recent generator proposed in [10], which we call **WorkflowHub**. Recall that both WorkflowGenerator and WorkflowHub are hand-crafted, while WfChef generators are automatically produced.

We consider workflow instances from four scientific applications: (i) Epigenomics, a bioinformatics workflow [28]; (ii) Montage, an astronomy workflow [29]; (iii) Cycles [30], an agriculture workflow; and (iv) Blast [31], a bioinformatics workflow.

[4] `https://github.com/wfcommons/wfcommons`

[5] `https://github.com/wfcommons/workflow-schema`

We choose Montage and Epigenomics because they are well-known and widely used in production. Additionally, both WorkflowGenerator and WorkflowHub can generate synthetic workflow instances for these applications, making it possible to compare our approach to previous work. Note that both these previously proposed generators support several scientific workflow applications. However, the only ones they have in common are Epigenomics and Montage. Two other applications are supported by WorkflowGenerator and WorkflowHub, but the workflow structures generated by WorkflowGenerator do not match that seen in the real workflows. While it may be that the generator was inherently flawed, the most likely reason is that the applications have evolved while the generator has not been updated. The synthetic workflows it produces have structures very different from that seen in the workflows generated by the latest version of these applications, making comparisons to synthetic workflows generated by WorkflowHub or WfChef meaningless. Note that this need to manually update workflow instance generators to track updates in the application is precisely one of the motivations behind WfChef. That is, WfChef fully eliminates application version dependency by automatically generating generators and using real instances as a basis for synthetic workflows. Finally, despite not being supported by WorkflowGenerator, we included Cycles for its simpler structure when compared to Montage and Epigenomics, and Blast, a bioinformatics workflow, to have a larger set of representative applications.

Our ground truth consists of real Montage, Epigenomics, Blast, and Cycles workflow instances. These instances are publicly available on the WorkflowHub repository [10]. They were obtained based on logs of application executions with the Pegasus [32] and Makeflow [31] workflow management systems on the Chameleon academic cloud testbed [33]. Specifically, we consider 15 Montage workflow instances with between 60 and 9,807 tasks, 26 Epigenomics workflow instances with between 43 and 1,697 tasks, 24 Cycles workflow instances with between 69 and 6,545 tasks, and 15 Blast workflow instances with between 45 and 305 tasks.

We generate synthetic workflow instances with the same number of tasks as real workflow instances, so as to compare synthetic instances to real instances. Both WorkflowGenerator and WorkflowHub encode application-specific knowledge to produce synthetic workflow instances for any desired number of tasks, n. Instead, WfChef generators rely on training data, i.e., real workflow instances. We use a simple "training and testing" approach. That is, for generating a synthetic workflow instance with n tasks, we invoke WfChefRecipe with all real workflow instances with $< n$ tasks. For instance, say we want to use WfChef to generate an Epigenomics workflow with 127 tasks. We have real Epigenomics instances for 75, 121, and 127 tasks. We invoke WfChefRecipe with the 75- and 121-tasks instances to generate the recipe. We then invoke WfChefGenerate, passing to it this receipt and asking it to generate a 127-tasks instance. In Section 6, we evaluate how the accuracy of the generated workflows is impacted by reducing the number of real workflow instances used for training.

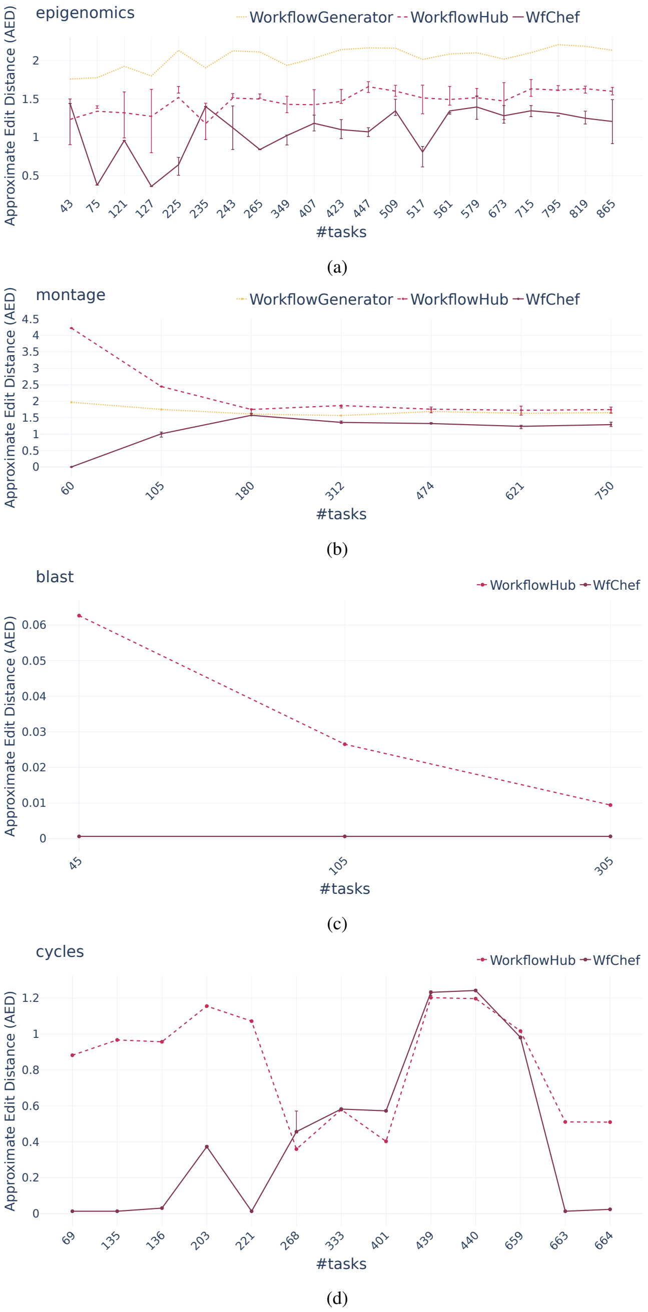
5.2. Evaluating the Realism of Synthetic Workflow Structures

We use two graph metrics to quantify the realism of generated workflows, as described hereafter.

Approximate Edit Distance (AED) – Given a real workflow instance w and a synthetic workflow instance w', the AED metric is computed as the approximate number of edits (vertex removal, vertex addition, edge removal, and edge addition) necessary so that $w = w'$, divided by $|w|$. Lower values include a higher similarity between w and w'. We compute this metric via the `optimize_graph_edit_distance` method from the Python's NetworkX package. Note that NetworkX also provides a method to compute an exact edit distance, but its complexity is prohibitive for the size of the workflow instances we consider. Even though the AED metric can be computed much faster, because it is approximate, we were able to compute it only for workflow instances with 865 or fewer tasks for Epigenomics, 750 or fewer tasks for Montage, 664 or fewer tasks for Cycles, and 45,105 or 305 tasks for Blast. This is because of RAM footprint issues (despite using a dedicated host with 192 GiB of RAM).

Figure 3 shows AED results for (a) Epigenomics, (b) Montage, (c) Blast, and (d) Cycles workflow instances, for WfChef, WorkflowGenerator, and WorkflowHub. (Note that results for Blast and Cycles are not shown for WorkflowGenerator as it cannot not produce workflow instances for these applications.) WorkflowHub and WfChef use randomization in their heuristics. Therefore, for each number of tasks we generated 10 sample synthetic workflows with each tool. The heights of the lines in Figure 3 correspond to average AED values, and we show error bars that represent the range between the third quartile (Q3) and the first quartile (Q1), in which 50 percent of the results lie. Error bars also show minimum and maximum values. Error bars, minimum, and maximum values are not shown for WorkflowGenerator as it generates synthetic workflow structures deterministically.

The key observation in Figure 3 is that WfChef leads to lower average AED values than its competitors in most cases. For Epigenomics, WorkflowGenerator leads to the worst results for all workflow sizes, being significantly outdone by WorkflowHub. WorkflowHub is itself outperformed by WfChef for all workflow sizes. On average, over all Epigenomics instances, WorkflowGenerator, WorkflowHub, and WfChef lead to an AED of 2.039, 1.473, and 1.086, respectively. For Montage workflows, WorkflowGenerator outperforms WorkflowHub for all instances, and both are outperformed by WfChef. On average, the AED over all Montage instances is 1.694 for WorkflowGenerator, 2.216 for WorkflowHub and 1.111 for WfChef. For Blast workflows, WfChef lead to a nearly zero AED, which is mostly due to the simple structure of the workflow, i.e., a single initial task which outputs are consumed by tasks representing a fan-out PO, which results are merged by two independent tasks. WorkflowHub, however, does not properly capture that larger instances of the workflow are defined by increasing the number of tasks in the fan-out PO. For Cycles workflows, both WfChef and WorkflowHub lead to similar AEDs for large instances, however WorkflowHub fails to capture work-



(a)

(b)

(c)

(d)

Figure 3: AED for (a) Epigenomics, (b) Montage, (c) Blast, and (d) Cycles workflows instances. Lines are average values. Error bars show the range between the third quartile (Q3) and the first quartile (Q1), and minimum and maximum values as black dots.

flow structure specifics for smaller instances. In this particular case, WorkflowHub attempts to generate more occurrences of a macroscopic PO instead of increasing the number of tasks in existing fan-in and fan-out patterns.

The good results obtained by WfChef are due to it being able to generate instances that are closer in size and that are more faithful to real workflow instances. Note that the AED metric values are quite high overall, often above 1 (except for Blast). Although the synthetic instances may have a structure that is overall similar to that of the real instances, making the two workflows absolutely identical requires a large number of edits. For this reason, hereafter we present results for a second metric.

Type Hash Frequency (THF) – Given a real workflow instance w and a synthetic workflow instance w', the THF metric is computed as the Root Mean Square Error (RMSE) of the frequencies of vertex type hashes. Recall from Definition 1 that the type hash of a vertex encodes information about a vertex's type but also the types of its ancestors and successors. Therefore, the more similar the workflow structure and sub-structures, the lower the THF metric.

Figure 4 shows THF results for (a) Epigenomics, (b) Montage, (c) Blast, and (d) Cycles. More results are shown than in Figure 3 since we can evaluate the THF metric for larger workflow instances. Like in Figure 3, lines represent averages and error bars, minimum, and maximum values are shown for WorkflowHub and WfChef.

Results are mostly in line with AED results. For Epigenomics, WorkflowGenerator leads to the worst average results for all workflow sizes. WfChef leads to significantly better results on average than WorkflowHub in all but two cases. For 349- and 423-task workflows, although WfChef leads to better average results, error bars for WfChef and WorkflowHub have a large amount of overlap. Note that the length of the error bars for the WfChef results show a fair amount of variation, with short error bars for one workflow size and significantly longer error bars for the next size up (e.g., going from 265 tasks to 349 tasks). This behavior is due to "jumps" in structure between workflows of certain scales. In other words, for a given application, it is common for smaller workflows to contain only a subset of the patterns that occur in larger workflows. On average over all Epigenomics instances, WorkflowGenerator, WorkflowHub, and WfChef lead to a THF of 0.097, 0.021, and 0.004, respectively. For Montage, WorkflowGenerator leads to better average results than WorkflowHub for all workflow sizes, and WfChef leads to strictly better results than its competitors for all workflow sizes. On average over all Montage instances, WorkflowGenerator, WorkflowHub, and WfChef lead to a THF of 0.211, 0.252, and 0.040, respectively. For Blast, WfChef leads to nearly zero THF values, though WorkflowHub also leads to very small values. For Cycles, WfChef significantly outperforms WorkflowHub, with average THF of 0.01 and 0.15, respectively. This indicates that WfChef can capture most patterns expressed in Cycles workflows.

We conclude that generators produced by WfChef generate synthetic workflow instances with structures that are significantly more realistic than that of workflows generated by WorkflowGenerator and WorkflowHub.

5.3. *Evaluating the Accuracy of Synthetic Workflows*

Synthetic workflow instances are typically used in the literature to drive simulations of workflow executions. A pragmatic way to evaluate the realism of synthetic workflow instances is thus to quantify the discrepancy between their simulated executions to that of their real counterparts, for executions simulated

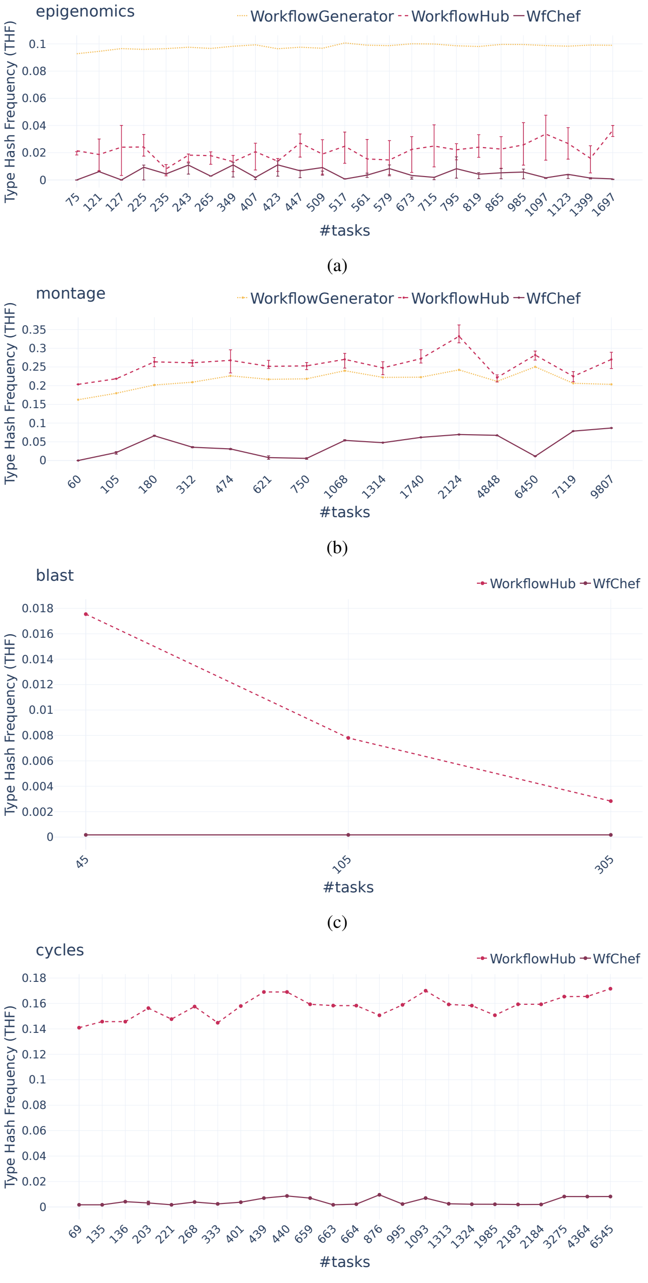


Figure 4: THF for (a) Epigenomics, (b) Montage, (c) Blast, and (d) Cycles workflows instances. Lines represent average values. Error bars show the range between the third quartile (Q3) and the first quartile (Q1), and minimum and maximum values as black dots.

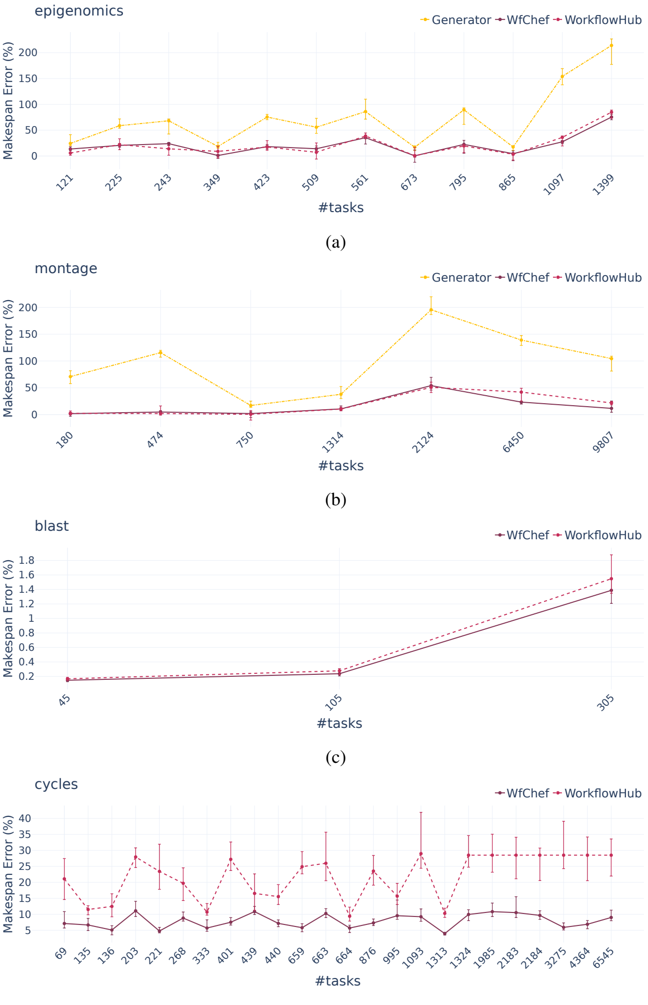


Figure 5: Makespan error for Epigenomics (*a*), Montage (*b*), Blast (*c*), and Cycles (*d*) workflows instances. Points are average values. Error bars show the range between the third quartile (Q3) and the first quartile (Q1), and minimum and maximum values as black dots.

for the same compute platform using the same Workflow Management System (WMS). To do so, we use simulators [34, 35] of the state-of-the-art Pegasus [32] and Makeflow [31] WMSs. The simulators are built using WRENCH [36], a framework for implementing simulators of WMSs that are accurate and can run scalably on a single computer. In [37], it was demonstrated that WRENCH provides high simulation accuracy for workflow executions using Pegasus and Makeflow. To ensure accurate and coherent comparisons, all simulation results in this section are obtained for the same simulated platform specification as that of the real-world platforms that was used to obtain the real workflow instances (based on execution logs): 4 compute nodes each with 48 cores on the Chameleon testbed [33].

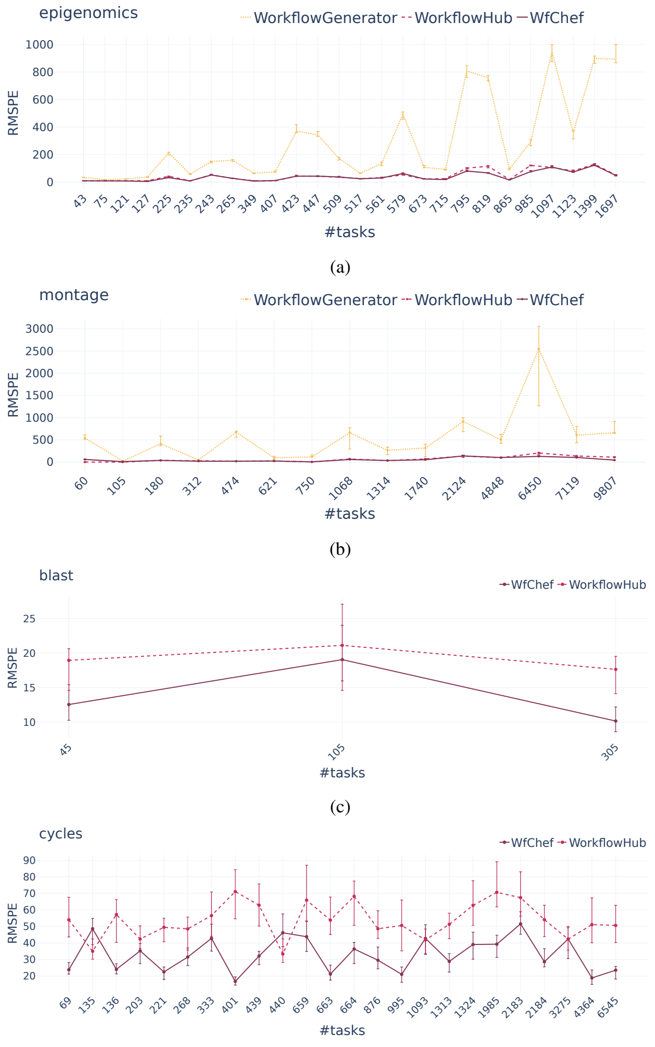
We quantify the discrepancies between the simulated execution of a synthetic workflow instance and that of a real workflow instance with the same number of vertices, using two metrics. The first metric is the absolute relative difference between the simulated makespans (i.e., overall execution times in seconds), which call makespan error. The second metric is the Root Mean Square Percentage Error (RMSPE) of workflow task start dates. The former metric is simpler (and used often in the literature to quantify simulation error), but the latter captures more detailed information about the temporal structure of the simulated executions.

Figure 5 shows makespan error for synthetic instances generated by WorkflowGenerator, WorkflowHub, and WfChef, for (a) Epigenomics, (b) Montage, (c) Blast, and (d) Cycles. Note that, unlike for the results in the previous section, error values are shown for WorkflowGenerator. Although it generates workflows with deterministic structure, it samples task characteristics (i.e., task runtimes, input/output data sizes) from particular random distribution. Both WorkflowHub and WfChef do a similar sampling, but from distributions determined via statistical analysis of real workflow instances.

Overall, the execution of synthetic workflows generated by WorkflowGenerator yield the least accurate makespans. WorkflowHub and WfChef lead to better results, with a small advantage for WorkflowHub for Epigenomics and Montage workflows, a small advantage for WfChef for Blast, and a relatively large advantage for WfChef for Cycles. Over all Epigenomics instances the average relative differences between makespans of the real workflow instances and of the synthetic instances generated by WorkflowGenerator, WorkflowHub, and WfChef are 75.73%, 15.21%, and 15.50%, respectively. For Montage instances, these averages are 135.12%, 32.61%, and 25.59%. For Blast instances, these averages are 0.69% and 0.60% for WorkflowHub and WfChef, respectively. Finally, the average relative differences between makespans for Cycles instances are 22.05% and 8.26%.

Figure 6 shows results for the RMSPE of workflow task start dates. Here again, we find that the synthetic workflow instances generated by WorkflowGenerator lead to unrealistic simulated execution. WorkflowHub and WfChef lead to more similar results, with a small advantage for WfChef (for Montage and Blast), and significant advantage for WfChef for Cycles. On average over all Epigenomics instances, the RMSPE of workflow task completion dates for synthetic Epigenomics instances generated by WorkflowGenerator, WorkflowHub, and WfChef are 294.70%, 46.08%, and 40.49%, respectively. For Montage instances, these averages are 558.93%, 64.29%, and 55.42%. For Blast instances, these averages are 18.94% and 13.63% for WorkflowHub and WfChef, respectively. For Cycles, these averages are 53.03% and 32.27%. The substantial difference for Cycles is mostly due to task graph patterns that are not captured by WorkflowHub.

We conclude that WfChef generators produce synthetic workflow instances that lead to simulated executions that are drastically more realistic than that of synthetic workflows generated by WorkflowGenerator. In fact, it is fair to say that WorkflowGenerator does not make it possible to obtain realistic simulation results (which is a concern given its popularity and commonplace use in the literature). WfChef generators lead to results that are similar but typically more accurate than WorkflowHub. And yet, WfChef generators are automatically generated meaning that, and very much unlike WorkflowGenerator and WorkflowHub, they require zero implementation effort.



(a)

(b)

(c)

(d)

Figure 6: RMSPE of simulated task start dates for Epigenomics (*a*) and Montage (*b*), Blast (*c*) and Cycles (*d*) workflows instances. Points are average values. Error bars show the range between the third quartile (Q3) and the first quartile (Q1), and minimum and maximum values as black dots.

6. Impact of the Real Instance Dataset Size

In the previous section, we have shown that WfChef has the ability to automate the detection of workflow sub-structures to create realistic synthetic workflow instances. Our approach requires the availability of real workflow instances. To date, a limited number of real-world scientific workflow instances are available, and only for a small number of workflow applications. For four of these applications, we have shown that WfChef can achieve good results. The dearth of available real workflow instance may be the reason why no machine learning algorithm has been proposed to solve the synthetic workflow generation problem — there simply is not enough data to learn from. It would be natural to expect that WfChef, like machine learning algorithms, would yield significantly better results with a greater number of available real instances. Yet, we have observed that in many cases using a larger number of real instances yields little to no improvements. This is likely because adding real instances that do not include new workflow patterns does not benefit WfChef, and in fact merely increases its computational complexity. It is thus possible that WfChef could achieve good results even when using a small number of real instances as its input.

In this section, we perform experiments to assess the realism of generated synthetic workflow instances when real instance sets of different sizes are used. We measure THF values

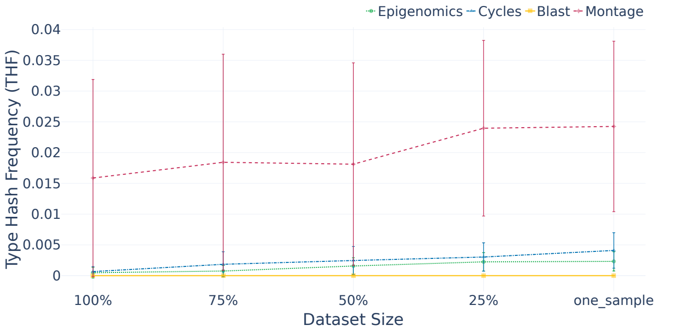


Figure 7: Type Hash Frequency (THF) vs. fraction of real workflow instances provided as input to WfChef.

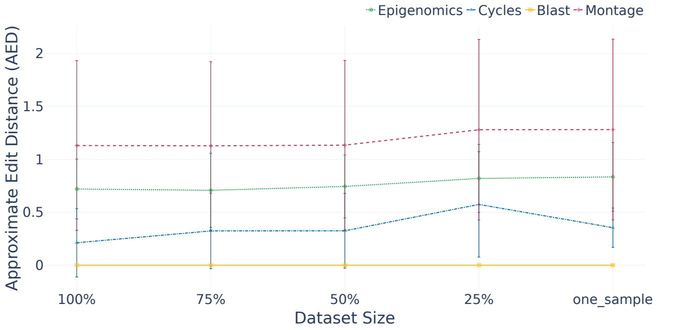


Figure 8: Approximate Edit Distance (AED) vs. fraction of real workflow instances provided as input to WfChef.

for the generated instances as well as the time required to generate them. The goal of these experiments is to better understand the impact of additional, non-informative workflow instances on the complexity of WfChef, and to draw conclusions regarding how large the real workflow instance dataset needs to be for WfChef to produce realistic synthetic workflow instances.

6.1. Accuracy vs. Dataset Size

In Section 5, we evaluate the realism of WfChef-generated synthetic instances. All experiments in that section are done using all of the real-world instances available to us for the Epigenomics, Montage, Blast, and Cycles applications. We wish to determine whether using these instances is truly needed to produce realistic synthetic workflow instances; and, if not, how many instances is a good number to use. To this end, here we run experiments in which we remove real instances and run WfChef to generate synthetic instances. Specifically, we use 100%, 75%, 50%, 25%, and a single one of the real instances, where the largest instances (in number of tasks) are removed. We then evaluate the generated synthetic instances using the same metrics as used in Section 5.2, Type Hash Frequency (THF), and Approximate Edit Distance (AED), for our four applications.

Figure 7 shows THF values of WfChef-generated workflow instances vs. the fraction of real instances used as input. For all four applications, using 100% of the real instances leads to the lowest THF values, but we note that using fewer instances does not lead to much higher THF values. For Montage and Cycles, the difference between the best result (achieved using 100% of the real instances) and the worst result (achieved when using a single real instance) is 0.0084 and 0.0034, for Montage and Cycles, respectively. For Epigenomics, the difference is 0.0018 between using 100% of the real instances and using 25% of them (which leads to the worst THF value). Due to the simple structure of Blast workflows, WfChef is able to achieves zero THF regardless of the number of real instances used.

Figure 8 shows AED values vs. the fraction of real instances used as input. The main observation is that AED values follows the same trends as THF values. For the Montage, Epigenomics, and Cycles applications the best results are achieved when using 100% of the real instances and the worst results are achieved when using 25% or a single one of the real instances. The difference between the minimum and the maximum AED values is 0.154, 0.127, and 0.362 for these three workflows, respectively. Finally, just as for THF, AED for Blast is also 0.000.

The results in Figure 7 and Figure 8 lead to the conclusion that using 100% of the real instances is not necessary to achieve good accuracy results. In fact, using 50% of the real instances yields results that are very close to the best results. This suggests that using a small number of real instances as input to WfChef is sufficient to produce realistic synthetic instances. We also note that the difference between the best and the worst results is not significant for any of the applications, which also supports the previous conclusion.

6.2. Recipe Creation Time vs. Dataset Size

As expected, Section 6.1 shows that using all real workflow instances available yields the most realistic synthetic instances for all our four applications. As seen in Section 4.2, however, creating recipes, which are then used to generate synthetic instances, requires applying three algorithms to each real instance. When these instances are large this process can be time-consuming.

In this section, we measure the time to create a recipe when using 100% or 50% of the available workflow instances. Recall from Section 6.1 that the larger instances are removed. That is, when using 50% of the real instances, these instances are the smallest ones. We thus expect large speedup when going from 100% to 50% if there are some very large instances in the dataset.

Table 1 shows recipe creation times for our four applications. In Section 6.1, using 50% of the available samples yields realistic synthetic instances that differ by at most 0.018 THF from the real workflows. The results in Table 1 show that creating the recipes with 50% of the samples can have large speedup (859.5 for Montage, 504.2 for Cycles). As expected, this dramatic speedup is due to the exclusion of the largest and most complex real instances, which, despite their complexity, do not contain much more new information about the workflow structure. For applications such as Epigenomics, which does not have very large instances, the speedup is only 1.5, and for Blast, which has simple structure and small instances, the speed up is minimal at 1.1.

The results in this section paired with the results from Section 6.1 show that using 50% of the real instances is sufficient

Application	Order of Samples Available	100% of instances	50% of instances
Montage	60, 105, 180, 312, 474, 621, 750, 1068, 1314, 1740, 2124, 4848, 6450, 7119, 9807	58,020.6s	67.5s
Epigenomics	43, 75, 121, 127, 225, 235, 243, 265, 349, 407, 423, 447, 509, 517, 561, 579, 673, 715, 795, 819, 865, 985, 1097, 1123, 1399, 1697	30.9s	21.2s
Cycles	69, 135, 136, 203, 221, 268, 333, 401, 439, 440, 659, 663, 664, 876, 995, 1093, 1313, 1324, 1985, 2183, 2184, 3275, 4364, 6545	45,728.2s	90.7s
Blast	45, 45, 45, 45, 45, 105, 105, 105, 105, 105, 305, 305, 305, 305, 305	19.5s	17.9s

Table 1: Recipe creation times when using 100% and 50% of the real instances for the Montage, Epigenomics, Cycles, and Blast applications.

to create sufficiently realistic synthetic instances, which also achieves a significant speedup over using 100% of the real instances.

7. Conclusion

The availability of synthetic but realistic scientific workflow instances is crucial for supporting research and development activities in the area of workflow computing, and in particular for evaluating workflow algorithms and systems. Although synthetic workflow instance generators have been developed in previous work, these generators were hand-crafted using expert knowledge of scientific applications. As a result, their development is labor-intensive and cannot easily scale to supporting large number of scientific applications. As an alternative, in this work we have presented WfChef, a tool for automatically generating generators of realistic synthetic scientific workflow instances. Given a set of real workflow instances for a particular scientific application, WfChef analyzes these instances in order to discover application-specific patterns. A synthetic workflow instance with an (almost) arbitrary number of tasks can then be generated by replicating these patterns in a real workflow instance with fewer tasks. We have demonstrated that the WfChef generators, which require zero software development efforts, generate more realistic synthetic workflow instances than the previously available hand-crafted generators. We have quantified workflow instance realism both based on workflow DAG metrics and on simulated workflow executions. We have also presented results that show how the size of the real instance dataset impacts WfChef. To do so we have measured the realism of the generated workflow instances when removing increasingly large numbers of instances from the dataset (from largest to smallest). Finally, we have shown that although using the full dataset yields better synthetic instances, using only 50% of the dataset produces instances that are almost as realistic and can be generated much quicker.

A short-term future work direction is to develop a metric for structurally comparing two workflow applications to reason about how one application might perform on some system, given its similarity to another workflow application. A longer-term direction is to investigate whether machine learning techniques can be applied to solve the synthetic workflow generation problem, to compare these techniques to WfChef, and perhaps evolve WfChef accordingly. Our suspicion, however, is that the amount of training data necessary for machine learning approaches to be effective could be prohibitive. By contrast, the WfChef algorithms are able to analyze a few real workflow instances to discover patterns and, as seen in Section 6, still yield good results.

Acknowledgments

This work is funded by NSF contracts #1923539,#1923621; partly funded by NSF contracts #2016610,#2016619,#2103489, #2103508. 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. We also thank the NSF Chameleon Cloud for providing time grants to access their resources.

References

[1] R. M. Badia Sala, E. Ayguadé Parra, J. J. Labarta Mancho, Workflows for science: A challenge when facing the convergence of HPC and big data, Supercomputing frontiers and innovations 4 (1) (2017) 27–47. doi:10.14529/jsfi170102.

[2] C. S. Liew, M. P. Atkinson, M. Galea, T. F. Ang, P. Martin, J. I. V. Hemert, Scientific workflows: moving across paradigms, ACM Computing Surveys (CSUR) 49 (4) (2016) 1–39. doi:10.1145/3012429.

[3] R. Ferreira da Silva, R. Filgueira, I. Pietri, M. Jiang, R. Sakellariou, E. Deelman, A characterization of workflow management systems for extreme-scale applications, Future Generation Computer Systems 75 (2017) 228–238. doi:10.1016/j.future.2017.02.026.

[4] Existing Workflow Systems, https://s.apache.org/existing-workflow-systems (2021).

[5] R. Ferreira da Silva, H. Casanova, K. Chard, I. Altintas, R. M. Badia, B. Balis, T. a. Coleman, F. Coppens, F. Di Natale, B. Enders, T. Fahringer, R. Filgueira, G. Fursin, D. Garijo, C. Goble, D. Howell, S. Jha, D. S. Katz, D. Laney, U. Leser, M. Malawski, K. Mehta, L. Pottier, J. Ozik, J. L. Peterson, L. Ramakrishnan, S. Soiland-Reyes, D. Thain, M. Wolf, A community roadmap for scientific workflows research and development, in: 2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS), 2021, pp. 81–90. doi:10.1109/WORKS54523.2021.00016.

[6] R. Ferreira da Silva, H. Casanova, K. Chard, T. Coleman, D. Laney, D. Ahn, S. Jha, D. Howell, S. Soiland-Reys, I. Altintas, D. Thain, R. Filgueira, et al., Workflows Community Summit: Advancing the State-of-the-art of Scientific Workflows Management Systems Research and Development (Jun. 2021). doi:10.5281/zenodo.4915801.

[7] R. Ferreira da Silva, H. Casanova, K. Chard, D. Laney, D. Ahn, S. Jha, et al., Workflows Community Summit: Bringing the Scientific Workflows Community Together (Mar. 2021). doi:10.5281/zenodo.4606958.

[8] R. Ferreira da Silva, R. M. Badia, V. Bala, D. Bard, T. Bremer, I. Buckley, S. Caino-Lores, K. Chard, C. Goble, S. Jha, D. S. Katz, D. Laney, M. Parashar, F. Suter, N. Tyler, T. Uram, I. Altintas, et al., Workflows Community Summit 2022: A Roadmap Revolution, Tech. Rep.

ORNL/TM-2023/2885, Oak Ridge National Laboratory (2023). doi:10.5281/zenodo.7750670.

[9] R. Ferreira da Silva, W. Chen, G. Juve, K. Vahi, E. Deelman, Community resources for enabling and evaluating research in distributed scientific workflows, in: 10th IEEE International Conference on e-Science, eScience'14, 2014, pp. 177–184. doi:10.1109/eScience.2014.44.

[10] R. Ferreira da Silva, L. Pottier, T. Coleman, E. Deelman, H. Casanova, Workflowhub: Community framework for enabling scientific workflow research and development, in: 2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS), 2020, pp. 49–56. doi:10.1109/WORKS51914.2020.00012.

[11] Wfinstances: Workflow Instances Repository, https://wfcommons.org/instances (2023).

[12] T. Coleman, H. Casanova, L. Pottier, M. Kaushik, E. Deelman, R. F. da Silva, Wfcommons: A framework for enabling scientific workflow research and development, Future Generation Computer Systems 128 (2022) 16–27. doi:10.1016/j.future.2021.09.043.

[13] J. Coleman, M. Kiamari, L. Clark, D. D'Souza, B. Krishnamachari, Graph convolutional network-based scheduler for distributing computation in the internet of robotic things, in: MILCOM 2022 - 2022 IEEE Military Communications Conference (MILCOM), 2022, pp. 1070–1075. doi:10.1109/MILCOM55135.2022.10017673.

[14] T. Coleman, H. Casanova, R. da Silva, Wfchef: Automated generation of accurate scientific workflow generators, in: 2021 IEEE 17th International Conference on eScience (eScience), IEEE Computer Society, Los Alamitos, CA, USA, 2021, pp. 159–168. doi:10.1109/eScience51609.2021.00026.
URL https://doi.ieeecomputersociety.org/10.1109/eScience51609.2021.00026

[15] DAGGEN: a synthetic task graph generator, https://github.com/frs69wq/daggen (2021).

[16] M. A. Amer, R. Lucas, Evaluating workflow tools with sdag, in: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, IEEE, 2012, pp. 54–63. doi:10.1109/SC.Companion.2012.20.

[17] D. G. Amalarethinam, 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 2 (3) (2011) 782.

[18] D. G. Amalarethinam, 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. doi:10.1007/978-3-642-30111-7_93.

[19] W. M. van Der Aalst, A. H. Ter Hofstede, B. Kiepuszewski, A. P. Barros, Workflow patterns, Distributed and parallel databases 14 (1) (2003) 5–51. doi:10.1023/A:1022883727209.

[20] U. Yildiz, A. Guabtni, A. H. Ngu, Towards scientific workflow patterns, in: Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science, 2009, pp. 1–10. doi:10.1145/1645164.1645177.

[21] D. Garijo, P. Alper, K. Belhajjame, O. Corcho, Y. Gil, C. Goble, Common motifs in scientific workflows: An empirical analysis, Future Generation Computer Systems 36 (2014) 338–351. doi:10.1016/j.future.2013.09.018.

[22] E. Slaughter, W. Wu, Y. Fu, L. Brandenburg, N. Garcia, W. Kautz, E. Marx, K. S. Morris, Q. Cao, G. Bosilca, S. Mirchandaney, W. Leek, S. Treichlerk, P. McCormick, A. Aiken, Task Bench: A Parameterized Benchmark for Evaluating Parallel Runtime Performance, in: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, 2020, pp. 1–15. doi:10.1109/SC41405.2020.00066.

[23] M. Roozmeh, I. Kondov, Workflow Generation with wfGenes, in: 2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS), 2020, pp. 9–16. doi:10.1109/WORKS51914.2020.00007.

[24] D. S. Katz, A. Merzky, Z. Zhang, S. Jha, Application skeletons: Construction and use in eScience, Future Generation Computer Systems 59 (2016) 114–124. doi:10.1016/j.future.2015.10.001.

[25] T. Coleman, H. Cansanova, K. Maheshwari, L. Pottier, S. R. Wilkinson, J. Wozniak, F. Suter, M. Shankar, R. Ferreira da Silva, Wf-Bench: 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. doi:10.1109/PMBS56514.2022.00014.

[26] J. Liu, E. Pacitti, P. Valduriez, M. Mattoso, A survey of data-intensive scientific workflow management, Journal of Grid Computing 13 (4) (2015) 457–493. doi:10.1007/s10723-015-9329-8.

[27] M. Malawski, A. Gajek, A. Zima, B. Balis, K. Figiela, Serverless execution of scientific workflows: Experiments with Hyperflow, AWS lambda and Google Cloud functions, Future Generation Computer Systems (2017). doi:10.1016/j.future.2017.10.029.

[28] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, K. Vahi, Characterizing and profiling scientific workflows, Future Generation Computer Systems 29 (3) (2013) 682–692. doi:10.1016/j.future.2012.08.015.

[29] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. Prince, et al., Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking, International Journal of Computational Science and Engineering 4 (2) (2009) 73–87. doi:10.1504/IJCSE.2009.026999.

[30] R. Ferreira da Silva, R. Mayani, Y. Shi, A. R. Kemanian, M. Rynge, E. Deelman, Empowering agroecosystem modeling with htc scientific workflows: The cycles model use case, in: 2019 IEEE International Conference on Big Data (Big Data), 2019, pp. 4545–4552. doi:10.1109/BigData47090.2019.9006107.

[31] M. Albrecht, P. Donnelly, P. Bui, D. Thain, Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids, in: Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies, 2012, pp. 1–13. doi:10.1145/2443416.2443417.

[32] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, K. Wenger, Pegasus, a workflow management system for science automation, Future Generation Computer Systems 46 (0) (2015) 17–35. doi:10.1016/j.future.2014.10.008.

[33] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, et al., Lessons learned from the Chameleon testbed, in: 2020 USENIX Annual Technical Conference (USENIX ATC 20), 2020, pp. 219–233.

[34] WRENCH Pegasus Simulator, https://github.com/wrench-project/pegasus (2021).

[35] WRENCH Makeflow Simulator, https://github.com/wrench-project/makeflow (2021).

[36] The WRENCH Project, http://wrench-project.org (2021).

[37] H. Casanova, R. Ferreira da Silva, R. Tanaka, S. Pandey, G. Jethwani, W. Koch, S. Albrecht, J. Oeth, F. Suter, Developing accurate and scalable simulators of production workflow management systems with wrench, Future Generation Computer Systems 112 (2020) 162–175. doi:10.1016/j.future.2020.05.030.