WfChef: Automated Generation of Accurate Scientific Workflow Generators

Tainã Coleman*, Henri Casanova†, Rafael Ferreira da Silva*
*Information Sciences Institute, University of Southern California, Marina Del Rey, CA, USA
†Information and Computer Sciences, University of Hawaii, Honolulu, HI, USA
{tcoleman,rafsilva}@isi.edu, henric@hawaii.edu

***Abstract*—Scientific workflow applications have become mainstream and their automated and efficient execution on large-scale compute platforms is the object 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. Dozens of workflow instances for a few scientific applications are available in public repositories. While these are invaluable, they are limited: workflow instances are not available for all application scales of interest. To address this limitation, previous work has developed generators of synthetic, but representative, workflow instances of arbitrary scales. These generators are popular, but implementing them is a manual, labor-intensive process that requires expert application knowledge. As a result, these generators only target a handful of applications, even though hundreds of applications use workflows in production.**

In this work, we present *WfChef*, a framework that fully automates the process of constructing a synthetic workflow generator for any scientific application. Based on an input set of workflow instances, WfChef automatically produces a synthetic workflow generator. We define and evaluate several metrics for quantifying the realism of the generated workflows. 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 the WfChef generators not only require zero development effort (because it is automatically produced), but also generate workflows that are more realistic than those generated by hand-crafted generators.

***Index Terms*—Scientific workflows, synthetic workflow generation, workflow management systems**

I. Introduction

Many computationally intensive scientific applications have been framed as *scientific workflows* that execute on various compute platforms and platform scales [1]. 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. 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[2] points to 290+ 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 [2], [3].

Addressing these challenges requires a solid experimental methodology for evaluating and benchmarking workflow algorithms and systems. A fundamental component of this methodology is the availability of sets of representative workflow instances. One approach is to infer workflow structures from real-world execution logs. We have ourselves followed this approach in previous work [4], [5], resulting in a repository that provides ∼20 workflow instances for each of a handful of scientific applications. These instances have been used by researchers, often for driving simulation experiments designed to evaluate scheduling and resource management algorithms.

Real workflow instances are by definition representative of real applications, but they cover only a limited number of scenarios. 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 [4] presented a synthetic workflow generator for four workflow applications, which has been used extensively by researchers[3]. 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 [5] 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 [5]. 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.

In this work, we present ***WfChef***, a framework that *automates* the process of constructing a synthetic workflow

[1]The IEEE Xplore digital database includes 118 articles with both the words "Workflow" and "Scheduling" in their title for 2020 alone.

[2]https://s.apache.org/existing-workflow-systems

[3]To date, 300+ bibliographical references to the research article and/or the software repository's URL.

generator for any given workflow application. WfChef takes as input a set of real workflow instances from an application, and outputs the code of a synthetic workflow generator for that application. WfChef analyzes the real workflow graphs in order to identify subgraphs that represent fundamental task dependency patterns. Based on the identified subgraphs and on measured task type frequencies in the real workflows, WfChef outputs a generator that can generate realistic synthetic workflow instances with an arbitrary numbers of tasks. In this work, we evaluate the realism of the synthetic workflows generated by our approach, both in terms of workflow structure and execution behavior. Specifically, this work makes the following contributions:

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 [4] and by the more recent generator in [5];
4) 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.

This paper is organized as follows. Section II discusses related work. Section III defines our target problem. Section IV describes WfChef. Section V presents experimental evaluation results. Finally, Section VI concludes with a summary of results and perspectives on future work.

II. 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 for various parameters [6]–[9]. For instance, DAGGEN [6] and SDAG [7] generate random DAGs based on ranges of 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 [8] generates random DAGS, but does so for parallel programs in which the task computation and communication payloads are modeled according to actual parallel programs. DAGITIZER [9] 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, they may not resemble those of 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 [10], 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 [11], 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 [12] focuses on identifying workflow "motifs" based on observing the data created and used by workflow tasks so as 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 [12]. In our previous work [4], 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 [5] and also results in Section V-C). In [5], 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 when compared to real-world workflows. 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.

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 for 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 [13], which can then be used to benchmark workflow systems.

III. Problem Statement

Consider a scientific application for which a list 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 [1], [2], [14], [15]). 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/exist 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 executable, and we simply define a vertex's type as the name of that 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 worfklows for most scientific applications cannot be feasibly instantiated for arbitray 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 V-B). 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 V-C).

IV. The WfChef Approach

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

A. 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 IV-A.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 shows an example for a simple 4-task diamond 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, resulting in different *pattern occurrences*, concept introduced in IV-A.2.

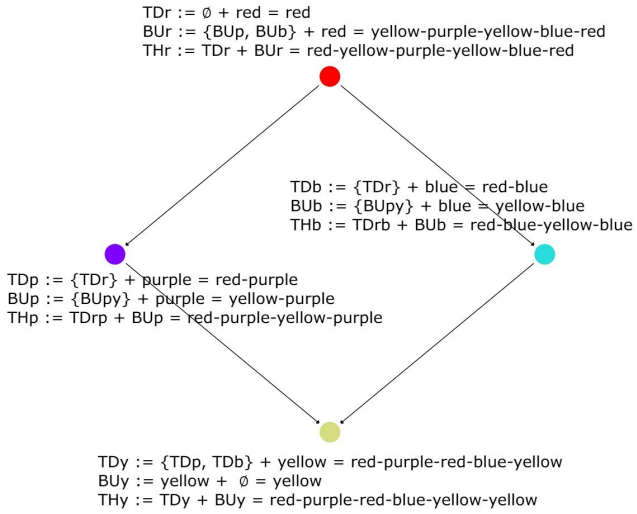

Fig. 1: Example workflow with TD, BU, and TH strings shown for each vertex. Vertex types are simply their colors ("red", "purple", "blue", or "yellow"). $\emptyset$ denotes the empty string, and $+$ denotes the string concatenation operator.

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 IV-A.2 (**Pattern Occurrence (PO)**):* Given W, a set of workflow instances for an application, 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.

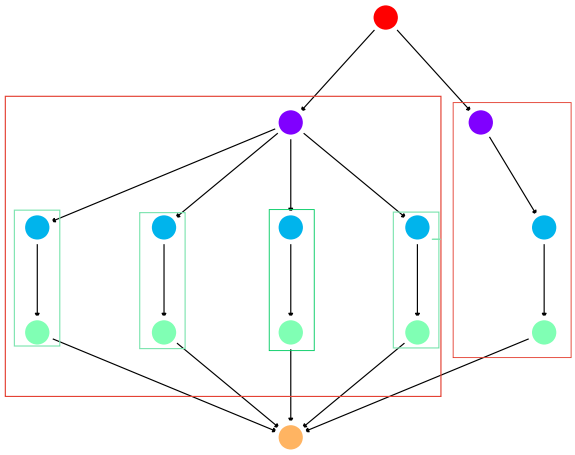

Fig. 2: Example workflow with 6 POs, shown in rectangular boxes. Boxes with the same color indicate POs with identical type hashes.

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.

B. 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.

The pseudo-code for WfChefRecipe 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 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).

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

```
1:  function WfChefRecipe(W)
2:      POs ← {}                                   ▷ dictionary of POs
3:      for each w ∈ W do
4:          POs[w] ← []                            ▷ 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 = ∅ or B = ∅ 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 ← {}                                ▷ dictionary of errors
18:     for each w ∈ W do
19:         for each b ∈ W s.t. |b| < |w| do
20:             g ← ReplicatePOs(|w|, b, POs[b], POs[w])
21:             Errors[b][w] ← Error(w,g)
22:         end for
23:     end for
24:     return new Recipe(W, POs, Errors)
25: end function
```

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 V-B). 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 ($Errors$).

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

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

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 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 ← base
3:     prob ← {}                    ▷ dictionary of probabilities
4:     for each po ∈ bPOs do
5:         nc = |{p ∈ cPOs | TH(p) = TH(po)}|
6:         tc = |{p ∈ cPOs}|
7:         nb = |{p ∈ bPOs | TH(p) = TH(po)}|
8:         prob[po] ← (nc/tc)/nb
9:     end for
10:    while |g| < n do
11:        po ← sample from bPO with distribution prob
12:        g ← 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).

C. Implementation

We have implemented our approach in a Python package called `wfchef`. Specifically, this package 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 [4]. The class has a public method `duplicate` that implements the WfChefGenerate algorithm, and a private method `duplicate_nodes` that implements the ReplicatePOs algorithm. This Python package is available on GitHub[5].

V. Experimental Evaluation

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

A. Methodology

We compare the realism of the synthetic workflow instances generated by WfChef generators to that of instances generated with the original workflow generator in [4], which is called **WorkflowGenerator**, and with the more recent generator proposed in [5], which is called **WorkflowHub**. Recall that both WorkflowGenerator and WorkflowHub are hand-crafted, while WfChef generators and automatically produced.

We consider workflow instances from two scientific applications: (i) Epigenomics, a bioinformatics workflow [16]; and

[4] https://github.com/wfcommons/workflow-schema
[5] https://github.com/tainagdcoleman/wfchef

(ii) Montage, an astronomy workflow [17]. We choose these two applications because they are well-known and used in production. But also, WorkflowGenerator and WorkflowHub can generate synthetic workflow instances for both of these applications, which makes 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 also by WorkflowHub. Unfortunately, WorkflowGenerator generates synthetic workflow structures that are no longer valid with respect to the current versions of these applications. Therefore, they are incorrect and not comparable to real workflow instances or to synthetic workflows generated by WorkflowHub.

Our ground truth consists of real Montage and Epigenomics workflow instances. These instances are publicly available on the WorkflowHub repository [5]. They were obtained based on logs of application executions with the Pegasus Workflow Management System [18] on the Chameleon academic cloud testbed [19]. Specifically, we consider 14 Montage workflow instances with between 105 and 9807 tasks, and 25 Epigenomics workflow instances with between 75 and 1697 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.

B. 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 and 750 or fewer tasks for Montage. This is because or RAM footprint issues (despite using a dedicated host with 192 GiB of RAM).

Figure 3 shows AED results for Epigenomics (top) and Montage (bottom) workflow instances, for WfChef, WorkflowGenerator, and WorkflowHub. WorkflowHub and WfChef use randomization in their heuristics. Therefore, for each number tasks we generated 10 sample synthetic workflow with each tool. The heights of the bars 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. Note that error bars, minimum, and maximum values are not shown for WorkflowGenerator as it generates synthetic workflow structures deterministically.

The key observation from Figure 3 is that in most cases WfChef leads to lower average AED values than its competitors. For Epigenomics, WorkflowGenerator leads to the worst results for all workflow sizes, being significantly outperformed 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 (Figure 3-*bottom*), WorkflowGenerator outperforms WorkflowHub for all workflow instances, and both are outperformed by WfChef. On average over all Montage instances, WorkflowGenerator, WorkflowHub, and WfChef lead to an AED of 1.694, 2.216, and 1.111, respectively.

The good results obtained by WfChef are due to it being able to generated 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. 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 IV-A.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 Epigenomics (top) and Montage (bottom). More results are shown than in Figure 3 since we can evaluate the THF metric for larger workflow instances. Like in Figure 3, bar heights 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

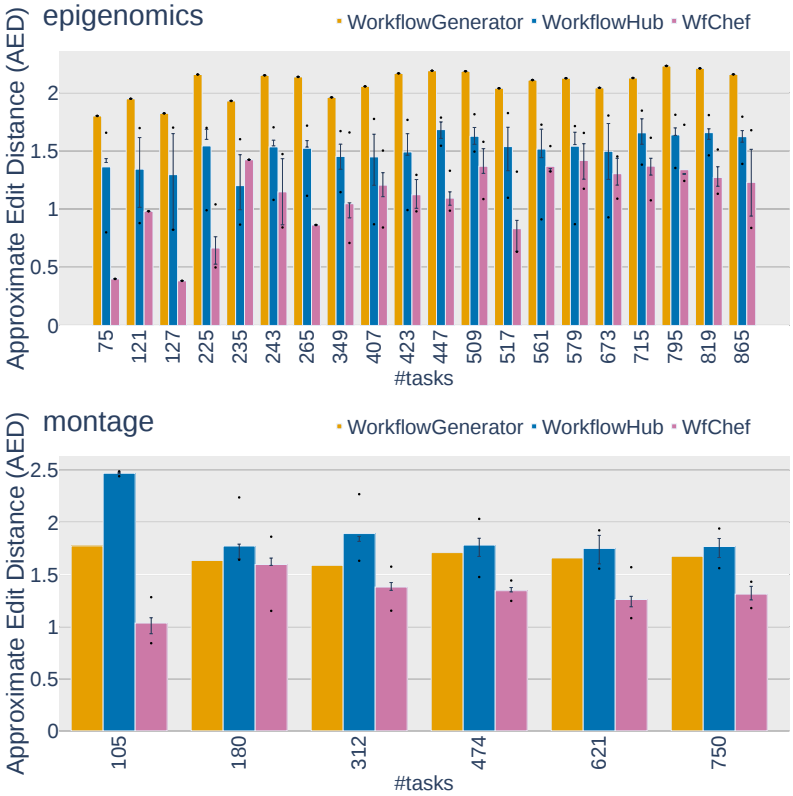

Fig. 3: AED for Epigenomics (*top*) and Montage (*bottom*) workflows instances. Bar heights are average values. Error bars show the range between the third quartile (Q3) and the first quartile (Q1), and minimum and maximum values as black dots.

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.

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.

C. 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 for the same compute platform using the same Workflow Management System (WMS). To do so, we use a simulator [20] of a the state-of-the-art Pegasus WMS [18]. The simulator is built using WRENCH [21], a framework for implementing simulators of WMSs that are accurate and can run scalably on a single computer. In [22], it was demonstrated that WRENCH provides high simulation accuracy for workflow executions using Pegasus. 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 processors on the Chameleon testbed [19].

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). 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

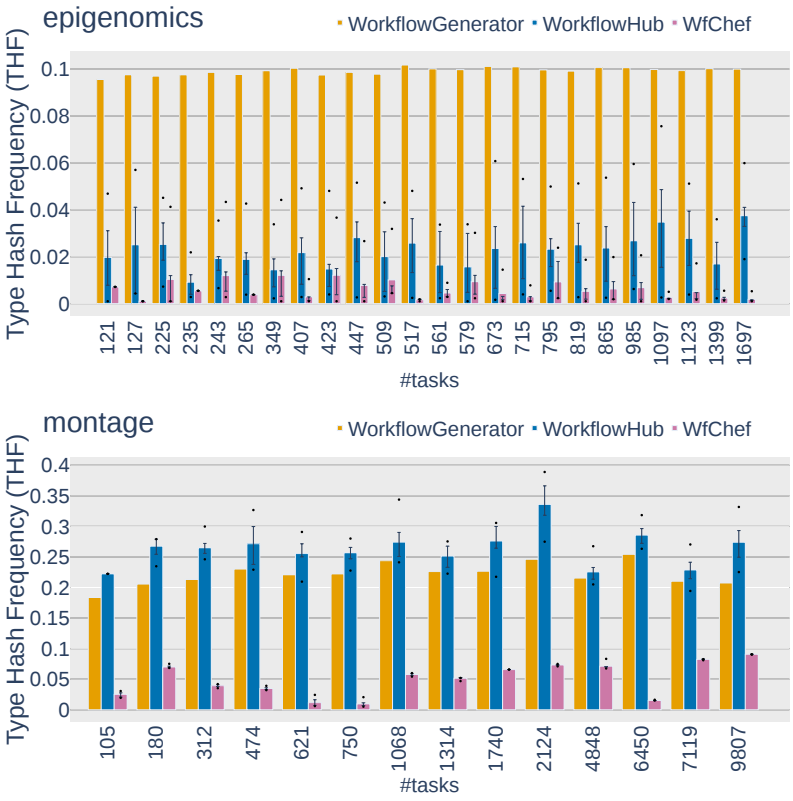

Fig. 4: THF for Epigenomics (*top*) and Montage (*bottom*) workflows instances. Bar heights are average values. Error bars show the range between the third quartile (Q3) and the first quartile (Q1), and minimum and maximum values as black dots.

executions.

Figure 5 shows makespans of simulated executions for real workflow instance and synthetic instances generated by WorkflowGenerator, WorkflowHub, and WfChef, for Epigenomics (top) and Montage (bottom). Note that, unlike for the results in the previous section, error bars 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, we find that the execution of synthetic workflows generated by WorkflowGenerator leads to the least accurate makespans. WorkflowHub and WfChef lead to better results, with a small advantage for WorkflowHub. On average 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%.

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 to WfChef. 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%.

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, require zero implementation effort.

VI. 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

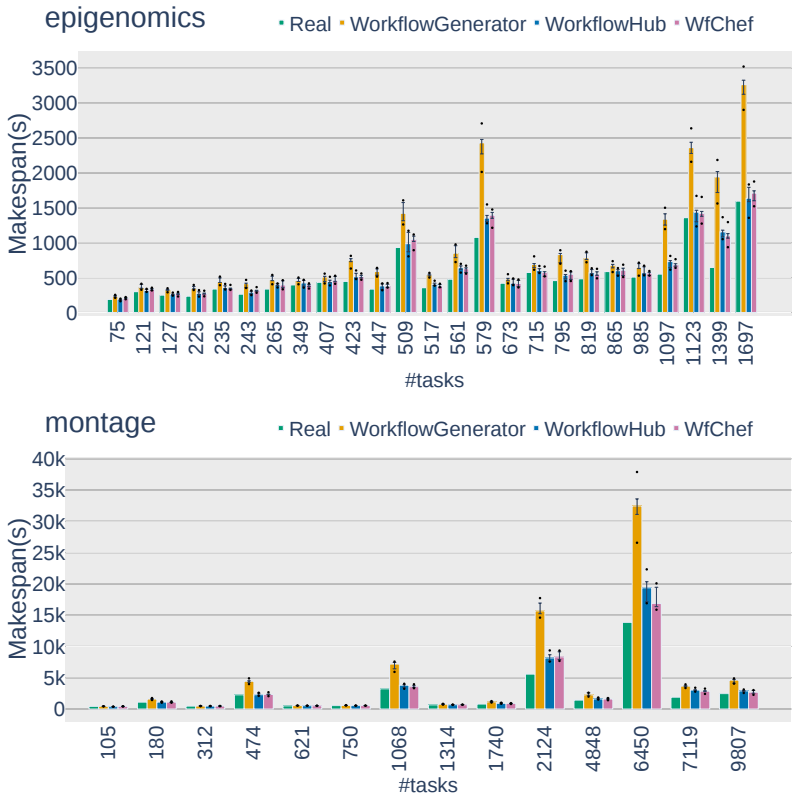

Fig. 5: Simulated makespan for Epigenomics (*top*) and Montage (*bottom*) workflows instances. Bar heights are average values. Error bars show the range between the third quartile (Q3) and the first quartile (Q1), and minimum and maximum values as black dots.

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 any 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 quantified workflow instance realism both based on workflow DAG metrics and on simulated workflow executions.

A short-term future work direction is to replace the hand-crafted WorkflowHub generators developed in [5] and available on the WorkflowHub web site[6] by generators automatically generated by WfChef. Another short-term future direction is to apply WfChef to more scientific workflow applications beyond those supported by WorkflowHub. 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. In fact, another interesting research direction is to determine the minimum amount of training data that still allows WfChef to produce realistic synthetic workflow instances. In the results presented in this work, for the purpose of evaluating WfChef and of comparing it to previously proposed approaches, we use as training data all available real workflow instances with fewer than the desired number of workflow tasks. But it may be that using fewer such instances would still lead to good results.

[6] https://workflowhub.org/generator

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

References

[1] 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.

[2] R. Ferreira da Silva, R. Filgueira, I. Pietri, M. Jiang, R. Sakellariou, and E. Deelman, "A characterization of workflow management systems for extreme-scale applications," *Future Generation Computer Systems*, vol. 75, pp. 228–238, 2017.

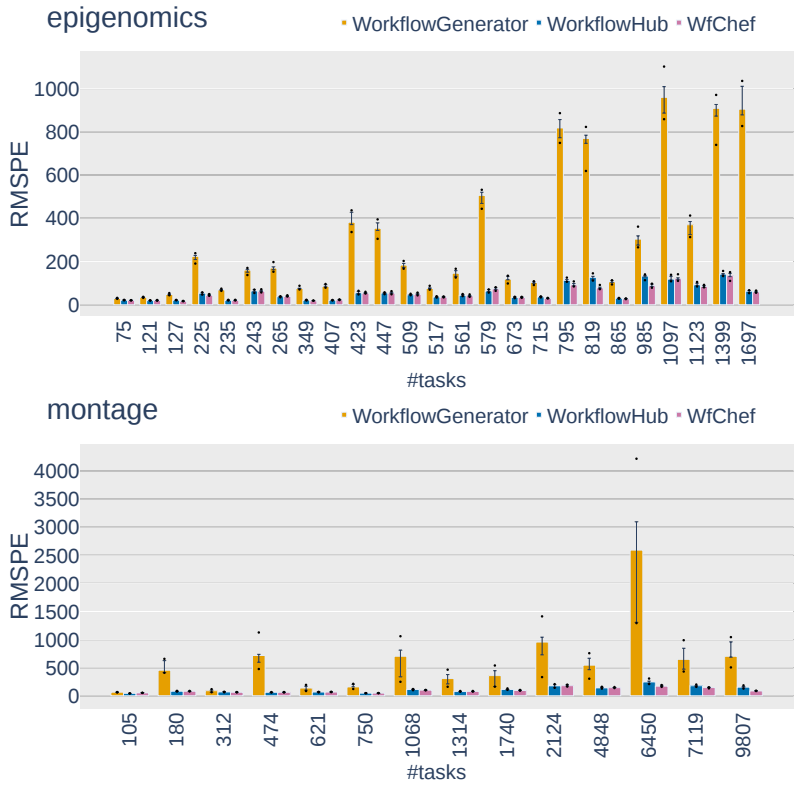

Fig. 6: RMSPE of simulated task start dates for Epigenomics (*top*) and Montage (*bottom*) workflows instances. Bar heights are average values. Error bars show the range between the third quartile (Q3) and the first quartile (Q1), and minimum and maximum values as black dots.

[3] 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.

[4] 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.

[5] R. Ferreira da Silva, L. Pottier, T. Coleman, E. Deelman, and 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.

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

[7] M. A. Amer and R. Lucas, "Evaluating workflow tools with sdag," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 2012, pp. 54–63.

[8] 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.

[9] 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.

[10] 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.

[11] U. Yildiz, A. Guabtni, and 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.

[12] 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.

[13] D. S. Katz, A. Merzky, Z. Zhang, and S. Jha, "Application skeletons: Construction and use in escience," *Future Generation Computer Systems*, vol. 59, pp. 114–124, 2016.

[14] J. Liu, E. Pacitti, P. Valduriez, and M. Mattoso, "A survey of data-intensive scientific workflow management," *Journal of Grid Computing*, vol. 13, no. 4, pp. 457–493, 2015.

[15] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions," *Future Generation Computer Systems*, 2017.

[16] G. Juve, A. Chervenak, E. Deelman, S. Bharathi, G. Mehta, and K. Vahi, "Characterizing and profiling scientific workflows," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 682–692, 2013.

[17] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. Prince *et al.*, "Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking," *International Journal of Computational Science and Engineering*, vol. 4, no. 2, pp. 73–87, 2009.

[18] 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, no. 0, pp. 17–35, 2015.

[19] 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.

[20] "WRENCH Pegasus Simulator," https://github.com/wrench-project/pegasus, 2021.

[21] "The WRENCH Project," http://wrench-project.org, 2021.

[22] H. Casanova, R. Ferreira da Silva, R. Tanaka, S. Pandey, G. Jethwani, W. Koch, S. Albrecht, J. Oeth, and F. Suter, "Developing accurate and scalable simulators of production workflow management systems with wrench," *Future Generation Computer Systems*, vol. 112, pp. 162–175, 2020.