Automatic Reproduction of Workflows in the Snakemake Workflow Catalog and nf-core Registries

Samuel Grayson
grayson5@illinois.edu
University of Illinois Urbana-Champaign
Urbana, Illinois, US

Darko Marinov
marinov@illinois.edu
University of Illinois Urbana-Champaign
Urbana, Illinois, US

Daniel S. Katz
d.katz@ieee.org
University of Illinois Urbana-Champaign
Urbana, Illinois, US

Reed Milewicz
rmilewi@sandia.gov
Sandia National Laboratories
Albuquerque, New Mexico, US

ABSTRACT

Workflows make it easier for scientists to assemble computational experiments consisting of many disparate components. However, those disparate components also increase the probability that the computational experiment fails to be reproducible. Even if software is reproducible today, it may become irreproducible tomorrow without the software changing at all, because of the constantly changing software environment in which the software is run.

To alleviate irreproducibility, workflow engines integrate with container engines. Additionally, communities that sprung up around workflow engines started to host registries for workflows that follow standards. These standards reduce the effort needed to make workflows automatically reproducible.

We study automatically reproducing workflows from two registries, focusing on non-crashing executions. The experimental data lets us analyze the upper bound to which workflow engines achieve reproducibility. We identify lessons learned in achieving reproducibility in practice.

CCS CONCEPTS

• **Software and its engineering** → **Software creation and management**; • **Information systems** → **Information systems applications**; • **Applied computing** → *Digital libraries and archives.*

KEYWORDS

reproducibility, computational experiments, research software engineering

ACM Reference Format:
Samuel Grayson, Darko Marinov, Daniel S. Katz, and Reed Milewicz. 2023. Automatic Reproduction of Workflows in the Snakemake Workflow Catalog and nf-core Registries. In *Proceedings of 2023 ACM Conference on Reproducibility and Replicability (ACM REP'23).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM REP'23, July 2023, Santa Cruz, California

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Scientific workflows have achieved prominence in recent years as a lingua franca for expressing computational experiments [14]. Workflows offer clarity of communication, potential for extensibility and reuse, and machine-readability that enables automated tooling. This success has led to a growing population of both workflows and workflow management systems on the open web [13].

Like all software, however, workflows can be irreproducible [17]; they may not be portable at all, or they may be portable at one point in time but decay later due to changes in computational environments [38]. Science is only self-correcting because scientists can scrutinize and build on each others' work [27], but irreproducibility hinders scientific progress. Scrutiny is hindered because readers cannot re-execute the workflow on their own computer. The communal practice of science is hindered because scientists cannot easily share workflows; they would have to independently re-develop each others' work.

Even outside of basic research, reproducibility of workflows is important. Suppose engineers use workflows to simulate the behavior of a physical part. Simulations are rapidly improving, so they may want to rerun a simulation done in the past with newer techniques or with different parameters. The physical part may have a lifetime measured in decades, but the software simulation is much more fragile, lasting only years. If the computation is not reproducible, engineers cannot easily rerun the simulation; they must either attempt time-consuming digital archaeology or rewrite the simulation from scratch.

A roadmap for workflow technologies by Deelman et al. notes an urgent need for innovative approaches, methods, and tools to ensure workflow reproducibility [7]. If properly archived and made discoverable, workflows could eventually become an enduring resource for the scientific community — enabling researchers to reproduce and build upon each others' work rapidly and credibly.

Having current data on the frequency and causes of workflow failures is key to building those solutions. A 2012 study by Zhao et al. examined failure causes among Taverna workflows from the myExperiment workflow registry [38]. Unfortunately, Taverna is not actively maintained. The landscape of workflow technologies has changed significantly, and Taverna has been replaced by newer tools (see Table 1). Public policy has also changed significantly since then, most notably in a memorandum from the US White House directing all federal research agencies to develop access plans for

their computational research results [28]. In short, several positive developments have happened on this front, and we need a refreshed perspective on workflow reproducibility.

To explore this topic further, we collected workflows from two workflow registries and attempted to reproduce crash-free executions for them. We address the following research questions:

- **RQ0.** What are the characteristics of workflows and revisions in the selected registries?
- **RQ1.** How many of the workflows (and for how many revisions of those workflows) in each selected registries are crash-free reproducible? This quantitatively assesses the level of reproducibility in practice for those registries.
- **RQ2.** For workflows that we were unable to reproduce crash-free executions, what are the most common failure modes? These modes inform future work of workflow engine developers for what to fix, researchers on automatic reproducibility of what to focus on inferring, and workflow users of what to watch out for.
- **RQ3.** What is the survival rate of crash-free reproducibility of workflows over time? While we cannot wait for a specific workflow to break, which may take months or years, we can assume that software in the future will behave similarly to software in the past, and make population-level inferences.
- **RQ4.** For crash-free reproductions, how much and what kinds of outputs are produced that are common between runs? Future research seeking to compare subsequent revisions in a semantic way will need to develop a handler for each kind of output. This research question tells them what kinds of outputs to focus on.

The main differences between our work and prior large-scale studies on automatic reproducibility [30, 36, 37, 37, 38] is:

- We study workflows not arbitrary computational experiments [30, 36, 37]. Workflows should stand a better chance for being reproduced because they natively use containers and are developed by scientists who should care more about reproducibility.
- We analyze the "survival rate" of workflows over time. To the best of our knowledge, prior work [36, 38] used time as a categorical rather than a continuous variable (informally "so many workflows from that year still work") or did not analyze time at all [30, 37].
- We analyze not only one but two registries and contrast their results. To our knowledge, prior work has not examined the similarities and differences between reproducibility from different workflow registries.

The remainder of this paper is structured as follows. Section 2 and 3 provide background and related work in the curation and sustainment of scientific workflows. Section 4 describes our data collection and analysis methodology. Section 5 presents the findings of our study. Section 6 provides a detailed discussion of those findings, and Section 6.1 outlines the limitations of our study. Finally, Section 7 summarizes the key results of our study and describes directions for future work.

2 BACKGROUND

The Association for Computing Machinery defines *reproducibility* and *replicability* as follows:

Reproducibility means "The measurement can be obtained with stated precision by a different team using the same measurement procedure, the same measuring system, under the same operating conditions, in the same or a different location on multiple trials. For computational experiments, this means that an independent group can obtain the same result using the author's own artifacts." [34]

Replicability means "The measurement can be obtained with stated precision by a different team, a different measuring system, in a different location on multiple trials. For computational experiments, this means that an independent group can obtain the same result using artifacts which they develop completely independently." [34]

For our study on reproducing scientific workflows, we define the following as "measurement":

Crash-free execution refers to whether the computational experiment runs to completion without crashing (specifically, terminating with a non-zero exit code for Unix programs).

While replicable research conclusions are the end goal, assessing that goal in practice requires expert case-by-case analysis. Assessing reproducible crash-free executions, on the other hand, is possible to do automatically and is a vital stepping-stone for replicable research conclusions. If an experiment has a reproducible crash-free execution, the workflow can be scrutinized, extended, and reused in future inquiry.

One salient question about reproducibility is how it relates to time. A computational experiment may be reproducible only up to some point in time but become irreproducible after that point. This change could be due for several reasons. For example, the software environment may not be fully specified, so retrieving the "latest" dependency may stop working at some point. It could also be because the software depends on some network resource which is no longer available. This phenomenon is often called **software collapse** [19], because software with an unstable foundation is analogous to a building with an unstable foundation. Software collapse for workflows manifests itself as irreproducible computational experiments.

3 PRIOR WORK

Prior work on large-scale quantitative reproducibility studies can be split into those whose reproduction is a automatic effort versus manual one.

Zhao et al. [38] automatically reproduce Taverna workflows from the myExperiment registry. However, Taverna is now defuncts and there have been many changes since then (see Table 1), so we should expect the results to change. Furthermore, Zhao et al. do not examine the correlation of failures with time or the kinds of outputs when the execution is successful.

Trisovic et al. [36] automatically run R code from the Harvard Dataverse repository. While Trisovic et al. do propose to study reproducibility based on R version and time (in their RQ8), they treat time as a categorical variable and do not perform a statistical

analysis to generalize their data. Furthermore, Trisovic et al.'s reproduction of R code does not know the order in which the scripts in a single project were originally run, so it incurs failures that may be simply due to a wrong order; our work studies workflows, which avoid the ordering problem because the workflow specifies dependencies between tasks.

Pimentel et al. [30] and Wang et al. [37] automatically run Jupyter Notebooks.automatically run Jupyter Notebooks from GitHub. Jupyter Notebooks have different strengths and use-cases than workflows. Jupyter Notebooks are usually used for at small, interactive jobs, whereas workflows are used for large, batch processing jobs. For example, Snakemake and Nextflow *at the language-level* both provide facilities to run jobs on a cluster. Snakemake and Nextflow *by default* write intermediate results to disk, so that workflows can be resumed if the node halts or needs to be restarted. While both batch-scheduling submission, crash-recovery, and containerization can be implemented in Python, workflow engines are more specialized to the use-case of analyzing data at a large scale. Therefore, we expect the reproducibility characteristics to be quite different. For example, Wang et al. find that using one set of Python packages, namely those in the default Anaconda repository, were sufficient for running their evaluation; workflows in Snakemake and Nextflow often provide a distinct set of Python packages *for each task*! This affects the reproducibility, as we will see in RQ2.

As an example manual reproduction, Krafczyk et al. execute an in-depth case study on a small set of computational experiments [23]. Stodden et al. [35] do case-studies with specific attention to journal policies. The case study methodology is useful for in-depth results but has difficulty in generalizing the results to an entire population. Our work attempts an automatic reproduction of a large set of experiments to address population-level questions.

Continuous integration [18] seeks to run tests at every change. However, software can fail not just by changes to the code itself, but also by changes to the environment (see "software collapse" above). Continuous integration usually does not seek to cover the case of a static code under an evolving environment. Beaulieu-Jones and Greene [4] propose "continuous analysis" to maintain reproducibility. The works are complementary; future work could look at combine techniques with this work to continuously test large-scale reproductions.

Provenance is also an important research direction. Pouchard et al. showed how collecting provenance data and performance metrics can aid in confirming the reproducibility of extreme-scale application workflows [31]; Meng and Thain developed a framework for capturing execution environments of workflows at a task-by-task level of granularity [26]. Large-scale reproduction tells provenance researchers where to start looking for examples of working workflows, examples of common errors, and other data. On the other hand provenance systems improve the reproducibility of workflow engines, which large-scale reproductions can evaluate.

Besides research literature, the community has been actively developing new policies, organizations, and tools to encourage reproducibility (see Table 1).

Table 1: A sample of tools, organizations, and policies regarding reproducibility since 2010.

Year	Kind	Description
2012	Tool	Snakemake paper [25]
2013	Policy	Geoscientific Model Development (GMD) journal requires code sharing [1]
2013	Policy	Office of Science and Technology Policy memorandum (Holden et al.) [20]
2015	Tool	Spack paper [16]
2015	Org	Volume 1 of ReScienceC published [33]
2017	Tool	Nextflow paper [10]
2017	Tool	Singularity paper [24]
2020	Org	Nextflow community curates nf-core [12]
2022	Policy	Office of Science and Technology Policy memorandum (Nelson et al.) [28]
2022	Policy	NASA Science Mission Directorate Science Policy Document 41a [39]

4 METHODOLOGY

The Workflow Community Initiative[1] lists four registries: Dockstore [29], Snakemake Workflow Catalog[2], WorkflowHub [15], and nf-core [12]. Dockstore and Workflowhub contain workflows of many different workflow languages, and they overlap as the same workflow can be in both registries. For this study, we chose Snakemake Workflow Catalog (SWC) and nf-core, since they contain only one workflow language, but are still well-populated. The Pegasus Workflow Engine also has a workflow hub called PegasusHub [3], but at the time of writing it only had twelve workflows, most of which were examples. Future work could extend our experimental prototype to multiple workflow registries.

Each entry in these registries refers to a specific project on GitHub. These registries are in machine-readable formats.

- The SWC registry includes any project on GitHub that has a `Snakefile` in the root directory or in `workflows`. Users can optionally include a `.snakemake-workflow-catalog.yml`, giving a machine-readable description of how to run their workflow. For these workflows we only attempt to reproduce revisions the developers mark as "releases" in GitHUb. These revisions are more likely to have worked on the developer's original machine.
- The nf-core registry is a community-curated set of analysis pipelines built using Nextflow. The authors follow conventions of having `main.nf` and `nextflow.config` files in the root directory. Also `nextflow.config` must define a profile for `test` and `docker`, `singularity`, `podman`, etc. These workflows have many users and contributors. Again, we only use git commits that the developers marked as releases.

Nextflow and Snakemake are **interpreters** for domain-specific language (DSL) that researchers use to write **workflow scripts**. The workflow scripts construct a DAG of **tasks** based on the input

[1] https://workflows.community/registries
[2] https://snakemake.github.io/snakemake-workflow-catalog/
[3] https://pegasushub.io/

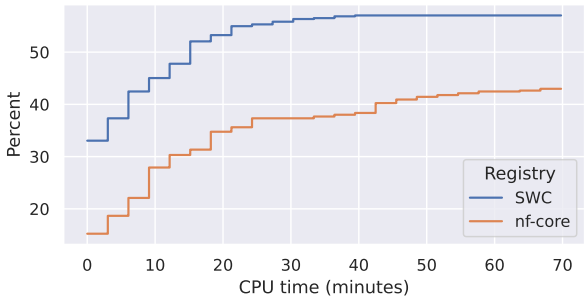

Figure 1: A cumulative frequency histogram of CPU time per revision (crashing and crash-free).

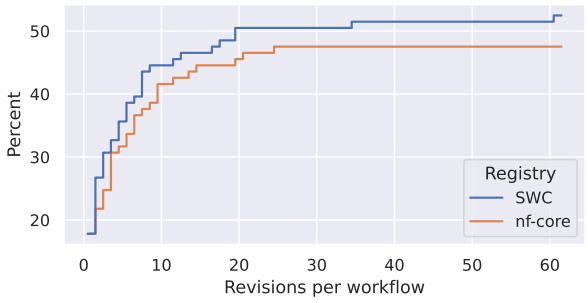

Figure 2: A cumulative frequency histogram of revisions per workflow in selected registries.

and configuration. The DSLs Nextflow and Snakemake contain directives to encapsulate the task in a specific **container**.

We use the appropriate workflow engine for each revision of each workflow in the registry. When we run the workflow, we are a different team using the same measuring system (experiment), therefore we are checking its reproducibility. Testing if the research result is consistent with a specific claim requires data from the original run and expert knowledge, so instead we just test if the default command with default parameters has a non-crashing execution. We run the experiments in a Spack environment[4] that has the workflow managers and their dependencies: Nextflow, Snakemake, Conda, Singularity, etc.

We also needed to install a few dependencies that workflows assume to exist on the system. Nextflow and Snakemake both allow the workflow to specify a container image to run the tasks in, but they do not provide a way to specify the environment of the program that generates the DAG. In some cases, that is done by convention, but these conventions are neither universally used nor automatically consumed by tools. These dependencies include domain-general processing tools, such as Numpy, Pandas, and Peppy. We discovered the exact set of dependencies through trial-and-error. We recognize that the computational experiments may rely on other unspecified dependencies or internet-accessible resources that no longer exist. We expect these to fail, and the object of this research is to count how many fail that way.

While most workflows finish within 30 minutes, some can take multiple hours (see Figure 1). In total, we spent over 5,600 CPU hours executing workflows. To reduce the waiting time, we use Parsl [3] to run different experiments on a parallel cluster. In our case, we used Microsoft Azure to provide the parallel cluster, but Parsl supports a wide range of parallel cluster providers or even a single node. Each job runs a specific revision of a project with a workflow engine (Snakemake or Nextflow), which can happen on any worker node in the cluster. Each worker node writes the output of their job to storage and sends the success indication back to the main node. At the end of execution, we have one or more "execution records" for every revision of every project.

[4] https://github.com/charmoniumQ/wf-reg-test/blob/main/spack/spack.yaml and https://github.com/charmoniumQ/wf-reg-test/blob/main/spack/spack.lock

We initially ran this process on a small random sample of revisions. Then we looked through every crash; some of them were due to the underlying workflow crashing, but some of them were artificial and actually caused by our testing infrastructure. We spent thirty minutes per crash debugging it; if there were no leads pointing to our infrastructure after that, we assumed the problem was with the workflow under test.

5 RESULTS

RQ0. We had to eliminate any workflows that had no published revisions; SWC has 2,045 workflows, but after eliminating the ones with no revisions, it only has 53. After this we came to 101 total workflows for both registries combined. Of these, we enumerated every revision that is considered a release, yielding 589. The number of releases per workflow follows a power-law distribution; 75% of the workflows from SWC have seven or fewer revisions, but one[5] has 61. The nf-core workflows are more evenly split, with 75% of the workflows having seven or fewer, and the maximum[6] having 25 revisions. Most workflows have just 1 release, but a prolific few have over 20 releases (see Figure 2).

Revisions of nf-core workflows are between 0 and 4.5 years old, while most revisions of SWC workflows are between 0 and 2 years old (see Figure 3). nf-core officially began in early 2018[7], while SWC only began in late 2020[8], but older revisions are possible. The registry only holds the URL of the GitHub repository, so a workflow with historical revisions stretching past the inception of the registry can be added to the registry, and our enumeration does include all those older revisions.

RQ0. The selected registries contain 101 workflows combined with 589 revisions, where the distribution of revisions to workflows follows a power-law distribution. The revisions are up to four years old, so **these registries are appropriate for analyzing mid-term reproducibility**.

[5] https://github.com/snakemake-workflows/dna-seq-varlociraptor
[6] https://github.com/nf-core/eager
[7] See https://nf-co.re/about
[8] See https://github.com/snakemake/snakemake-workflow-catalog/commit/4d0155429dc2fb75e4916e0e938d02f8b1efcb81

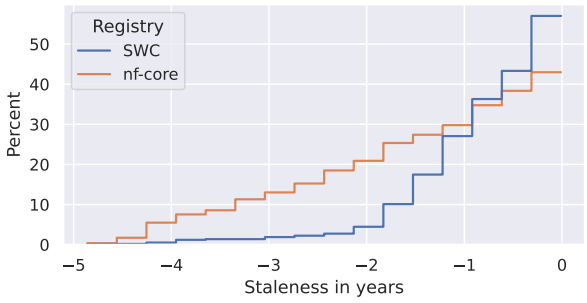

Figure 3: A cumulative frequency histogram of staleness of executions in selected registries.

RQ1. We were able to reproduce 28% of the total revisions from both registries. The nf-core workflows had a much higher crash-free reproducibility rate, 51%, compared to SWC workflows, 11%. This difference is surprising, considering that revisions of the nf-core workflows are older on average (see Figure 3). There are multiple explanations for this difference:

- The nf-core registry is a curated selection of workflows that select for popularly used pipelines.
- Of the workflows that we examined in more detail, the nf-core workflows defer most of their data processing to containerized tasks rather than the main script, which is not containerized, whereas SWC workflows do some data processing processing in the main script and some in containerized tasks. For example, some SWC workflows require BioPython to run the main workflow script.
- The SWC standard does not identify an obvious place for a default or example configuration. In RQ2, we will see that many of these failures are due to missing example data. The nf-core registry requires workflows to have a `test` profile, where this information can go.
- The Spack package manager interacts poorly with Conda.[9] This issue affects Snakemake workflows, which use Conda to manage environments, but not nf-core workflows which are less likely to use Conda environments.

Zhao et al. [38] find a 20% of crash-free reproduction. The SWC registry has a lower reproducibility rate, probably because our experimental infrastructure introduces conflicts between Spack and Conda (see above). The nf-core registry has a higher reproducibility rate, probably because it is community-curated, whereas Zhao et al. tested workflows in a self-depository called myExperiment. myExperiment is like Zenodo or GitHub, in that almost anyone can upload almost anything.

Trisovic et al. [36] in a recent study measured a success rate of 25% for R scripts, but they did not include timeout errors. When including timeouts as failures (the computational experiment might be reproducible, but we could not reproduce it), the rate fails to 12%, which is lower than nf-core but on par with SWC. We expect that workflows should be more reproducible than a bundle of R scripts, because repositories with R scripts may not specify the order in which to run the R scripts if there are dependencies between them; workflows encode that dependency explicitly. However, issues similar to the SWC environment are a possible culprit. It is also possible that the scripts studied by Trisovic et al. are from such different domains that the crash rate may be different.

[9]The problem, although dizzying, is as follows; We use Spack to install various Python packages needed for our experiment and Conda. Spack internally uses Pip to install those Python packages. Some computational experiment requests a Conda environment that includes Python packages. When Conda creates a new environment, it tries to reuse all of the packages installed by Pip. However, these versions of packages installed by Spack may conflict with those needed by Conda. Since these are installed by Pip, Conda cannot uninstall these.

Quantity	Total	nf-core	SWC
# workflows	101	48	53
# revisions	584	251	333
% of non-crashing revisions	28%	51%	11%
% of workflows with at least one non-crashing revision	53%	88%	23%

Table 2: Summary of data from automatic reproduction.

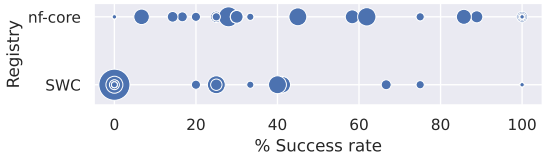

Figure 4: Each workflow is represented by a bubble: its radius is proportional to the number of revisions, its y-position corresponds to its registry, and its x-position is the percent of revisions that were reproducible.

For both registries we study, the successful executions do not all come from the same workflows. Namely, workflows are not simply reproducible in either most of their revisions or none; rather we see a diversity of success rates across workflows (see Figure 4).

The reproducibility of SWC is biased by many revisions coming from workflows where none of them can be reproduced, and nine of those are due to the same kind of crash. That one cause brings the number of non-crashing SWC workflow revisions down a bit. On the other hand, three of the working workflows have just one revision. The nf-core workflows, being more evenly spread in the number of revisions, do not suffer the same way.

RQ1. In all, we reproduced non-crashing executions for 28% of all revisions of all workflows in our selected registries. Considering the prevalence of irreproducibility, **more work needs to be done on achieveing reproducibility with low effort.**

RQ2. For each crashing execution, we examined the log files and standard error to find the low-level cause of the crash. Then we wrote a regular expression which could parse the information for other crashes of this cause (e.g., if the program crashed because an exception was thrown in a Snakemake script, we wrote a regular expression to parse the traceback). Next, we repeated the process

for the first crash not classified by the set of previously written regular expressions. Note that these crashes are only the *earliest* crash present in the code. If we were to fix the immediate crash, another crash of a different kind may still happen later, for which we have no information. Finally, we putatively categorized the failure cases according to their causes:

- **Missing input**: the crash was due to data or configuration.
- **Missing dependency**: the crash was due to missing a dependency.
- **Experiment error**: a crash caused by some aspect of the experimental testbench that would not be a problem for users outside of the testbench.
- **Network resource changed**: the workflow expects a network resource to have a different behavior than it currently has, e.g., a workflow queries a database using an API that has since changed.
- **Timeout reached**: we limited each revision to 2 hours.
- **Unclassified**: not all reasons for crashes can be easily identified automatically. For example, two workflows may fail with the same `IndexError`; in one case, it can be because the code is trying to access a configuration file that was not passed (i.e., missing input), but in another case the same textual error may be caused by a bug in the script (i.e., workflow script error).
- **Other (workflow script)**: the workflow fails for some other reason, and the crash happens within the workflow script, the program that generates a DAG of tasks.
- **Other (containerized task)**: the workflow fails for some other reason, and the crash happens within one of the containerized tasks, which are nodes of the workflow DAG.

These reasons are similar to those in Zhao et al. [38], but we allow for the timeout to be reached, and we allow "other." The "other" crashes indicate that the workflow was started correctly by our experimental testbench, had all necessary inputs, had complete software dependencies, and did not reach a timeout. Therefore, the "other" crashes are probably due to the workflow never working at all. This cause is consistent with what we find when we analyze those errors manually.

RQ2. Among workflows that crashed, the leading cause of failure was missing input such as input data or configuration files. The problem of missing is more prevalent in SWC because there is **no place in `.snakemake-workflow-catalog.yaml` to specify example/test data**.

RQ3. Unfortunately, the registries do not record any output[10] from the original run. While one cannot look back in time to see when any individual revision stopped working, one can instead reason about the aggregate population of revisions. We assume that the probability that a workflow published two years ago works is a good estimate for the probability that a workflow published today will work in two years. We attempt to find a trend between success rate and "staleness," difference in time between when the revision was published and when it is being executed, in 2023.

[10] See Section 6 for why GitHub CI is not sufficient for this purpose.

Kind of crash	all	nf-core	SWC
Missing input	32.2%	16.7%	43.8%
Missing dependency	13.7%	5.2%	20.1%
Unclassified reason	7.7%	2.0%	12.0%
Timeout reached	7.0%	8.8%	5.7%
Other (workflow script)	5.7%	11.2%	1.5%
Other (containerized task)	4.8%	2.8%	6.3%
Network resource changed	0.7%	1.6%	0.0%
Experiment error	0.2%	0.4%	0.0%
No crash	28.1%	51.4%	10.5%
Total	100%	100%	100%

Table 3: Reasons for crashes in revisions we failed to reproduce. These percentages are normalized to the *total* number of executions (crashing and crash-free).

	λ (95% CI)	k (95% CI)	Med. Survival Time
All Workflows	717 (657 – 777)	1.15 (1.06; 1.24)	521d
SWC	420 (382 – 458)	1.31 (1.20; 1.43)	317d
nf-core	1292 (1144 – 1439)	1.59 (1.36; 1.82)	1026d

We make the simplifying assumption that all workflows in our selected registries were reproducible when they were originally uploaded, and with each passing day there is a chance that a change will cause a workflow to break. We also assume that, as workflows age, the risk that they will break increases over time. To model this behavior, we fitted a Weibull survival function on our data. A Weibull probability distribution is described by the following parameterized formula:

$$f_X(x;\lambda,k) = \begin{cases} \frac{k}{\lambda}\left(\frac{x}{k}\right)^{k-1} \exp\left(-(x/\lambda)^k\right) & x \geq 0 \\ 0 & x < 0 \end{cases} \quad (1)$$

The parameter x represents the time-to-failure for a workflow. The parameter k is the shape parameter; a value of $k < 1$ means that failure rate decreases over time, $k = 1$ means that the failure rate is constant, and $k > 1$ means that the failure rate increases over time. Finally, λ, is a scale parameter; it can be interpreted as how much time must pass until 63.2% of workflows have failed. Using the Python package *lifelines* (version 0.27.4), we fit curves on our workflow reproducibility data for SWC workflows, nf-core workflows, and all workflows taken together. Below we provide a summary table of the curves, which are shown in Figure 5.

The difference in outcomes between the nf-core and SWC workflows calls attention to characteristics that we are not directly measuring. The nf-core registry is a carefully curated, community-driven effort to build and sustain nf-core genomics workflows, and most of the failing cases are old versions which are no longer officially supported. The SWC workflows, meanwhile, are drawn from a much larger corpus drawn from across GitHub. The fact that curated revisions of workflows survive three times longer than those

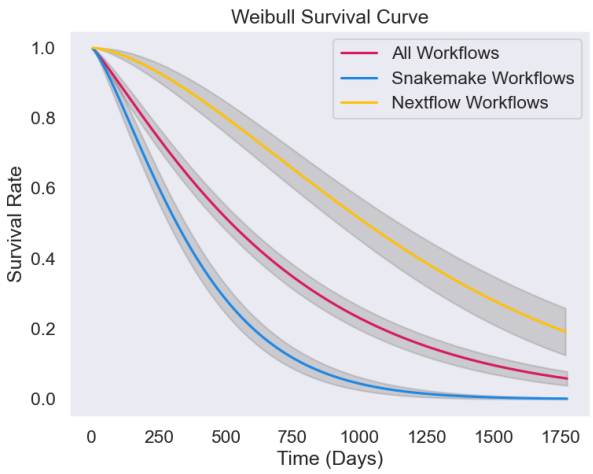

Figure 5: Estimates for expected workflow reproducibility over time modeled using a Weibull decay function. The estimated median survival time across all workflows in either registry is 1.42 years, which is to say that half of workflows remain crash-free and reproducible after that much time has passed. As the graph indicates, however, the nf-core workflows are much longer-lived (median 2.81 years) compared to the SWC workflows (median 0.81 years).

in the wild is not surprising, but it does show that a sharp decline in reproducibility is not inevitable.

RQ3. Aging workflows are more likely, all things being equal, to crash. However, different populations of workflows (such as those drawn from different registries) can **decay at different rates**. Moreover, **biological survival analysis is a useful tool** to study software collapse and plan ahead for it.

RQ4. This work just examines the reproducibility of crash-free executions. While reproducibility of research result requires expert knowledge, there may be intermediate levels of reproducibility that can be automatically assessed. An automated tool might look at the outputs produced, and if they have the same structure, compute the variation between. "Structure" here refers to the structure of the files (e.g., what is the directory and filename of specific output datasets) and the structure within the files (e.g., the order of columns in CSV). This tool would need to know what types of files are common outputs between the two executions.

We say a file path relative to the experiment's working directory is an **output** if it does not match a list of known intermediate outputs, log files, or temporary data. This is biased towards assuming a file is an output, since that is the default. The list of known non-output files includes `work/`, `pipeline_info`, `.nextflow.log`, `.nextflow` for Nextflow and `.snakemake` and `logs` for Snakemake. We also added directives to store the Singularity container file systems and Conda environments in locations, so they would not be considered outputs. Then we say an output is **common** to a workflow, if the workflow has more than two revisions with crash-free executions, and the output is present with the same file type in at least two thirds of the executions. We used `file`[11], a "file type guesser," to guess the file type.

Type	Total	nf-core	smk-wf-cat
ASCII text	85%	100%	33%
HTML document	59%	76%	0%
SVG image	26%	33%	0%
Zip archive data	7%	10%	0%
XML 1.0 document	4%	5%	0%
CSV text	19%	24%	0%
JSON text data	15%	19%	0%
very short file (no magic)	4%	5%	0%
gzip compressed data	11%	14%	0%
PDF document	7%	10%	0%
Blocked GNU Zip Format	7%	5%	17%
PNG image data	4%	5%	0%
LaTeX 2e document	4%	5%	0%
Total	27	21	6

Table 4: Each row shows what proportion of workflows with multiple revisions with crash-free executions have a common output of that file type. Note that these need not add to zero; one workflow might have a common output of ASCII text and another common output of PNG images; this would count as 1 in both rows.

The result (Table 4) shows that when workflows with multiple revisions with crash-free execution have common outputs, at least one of them usually an ASCII text, with no "higher level" structure like, CSV or JSON. Only 25% of the nf-core workflows can be compared through CSV. If one cannot deduce any structure to plain text files, the only choice may be to treat them as strings or list of strings (for each line). Future work may look investigate methods for quantifying the difference between ASCII text files, for example: one could use the edit distance at a line-level (this is what `diff` does). If there are a lot of sub-line changes, one might use edit-distance on characters, although this would be quite slow. If the sub-line changes are mostly numeric, one might treat the numbers as one token and compute the norm of the differences between the changes. If the files were of a fixed-length form, one could use Hamming distance.

RQ4. The most common output across revisions of a workflow is usually plaintext. Automatic reproducibility studies and provenance **researchers should research how to use plaintext to compare runs**.

6 LESSONS LEARNED

Superuser is required to reproduce normal workflows!

[11] See https://www.darwinsys.com/file/.

Since big supercomputers are often shared among a whole department or even multiple institutions, the users often do not have superuser access. Ideally, one should not need superuser access just to reproduce someone else's computational experiment. However, some functionality in container engines currently requires superuser, which did end up affecting us in this work. While Singularity and its successor Apptainer *can* be installed as a normal user, they do not support full set of features and experience worse filesystem performance than when installed as a superuser [9, 11] in "setuid mode." We noticed several failures due to inability to mount the right paths in a Singularity container, which we fixed by installing Singularity as superuser. While one does not need to run the workflow engine as a superuser, it calls Singularity, which calls a setuid binary that escalates into superuser privileges, so that binary has to be installed by a superuser. Also note that setuid Singularity cannot nest within another Singularity (setuid or not) [11], so we had to run our experiment on baremetal so that the workflow engine could start a setuid Singularity container.

Ongoing developments in the "user namespace" feature of the Linux Kernel open the door to container engines that do not run as root, but scientific users are often hindered by old versions of the kernel; CentOS 6 uses the 2.6 Linux Kernel, but user namespaces were not available until 3.8 or later [8]. Even in later kernels, user namespaces may be disabled. Enabling user namespaces opens a much larger surface for attacks (e.g., see CVE-2020-14386 in Linux Kernel 5.9 [6]), so many security standards recommend disabling them [2]. Still, Linux developers are making progress in securing user namespaces, and old supercomputers are being retired, so eventually, reproducibility can be improved through the use of rootless containers. Given a recent enough kernel, Charliecloud [32] provides exactly that. However, Snakemake has yet to integrate with Charliecloud (see ongoing issue[12]). Future work could quantify how the choice of container engine and root-user privilege changes the non-crashing reproducibility rate.

> The presence and rigor of community standards greatly affects reproducibility.

The nf-core repositories usually have a configuration profile in the root called `test` that runs whatever the workflow author defines as a test. Other tools choose conventions to make their tools easier to use (e.g., `mvn test`, `make all`, etc.). Nextflow workflows outside of nf-core do not usually follow this convention[13], so it would be much harder to automatically test them.

SWC does have a convention like this, but it is not rigorous enough. While SWC workflows have a place for "mandatory flags" in the `.snakemake-workflow-catalog.yml`, there is no place for "test data flags." As such, many of the workflows fail because the default command we use does not provide them any example data.

> CI scripts don't help much.

Often, a human could glean how to run a computational experiment given the CI scripts. However, selecting the right target is difficult to automate, because the CI scripts contain instructions for many different goals besides the goal of testing the software. When looking at GitHub Action scripts in SWC, we found scripts that lint, generate reports (without running), and test the conda environment; these would have to be excluded by automatic reproducibility software. If the full computational experiment has a long running time, users will exclude it from CI testing; there is no guarantee that any CI actions actually tests the code. This is another case where having a clear convention for naming the a CI action that tests the workflow would help.

> There can be more than one way to test the workflow.

The CI discussion also brings up another point, what should the "test configuration" be? Should it be a scaled down execution, or a full-fledged on? What if the experiment supports multiple different modes. Which should be used? In practice, the nf-core repositories specify one configuration as the "default" test configuration, but they often contain multiple `test_*` configurations, providing for test variants. An open ontology could describe what knobs there are to turn in each test. This opens the door to a lot of automatic testing applications, such as autotuning configuration parameters, outcome-preserving input reduction, and other kinds of parameter searching, if the system knows what knobs it is allowed to turn without breaking the semantics of the experiment.

> Workflow authors should report resource requirements in a machine-readable format.

Each of these different test variants may have a different resource requirement too. In batch compute systems, such as supercomputers, on which many computational scientists work, the users request a compute resource allocation (often number of CPUs, number of GPUs, peak disk utilization, peak memory utilization, and total time). In practice, this is guessed using rules-of-thumb; if the guess is wrong, their job may fail, and they will have to retry with a larger resource request. In fact for this very work, we submitted a proposal with very inaccurate back-of-the-envelope estimates. When we got an allocation, we did not know what value to set as the timeout. While not strictly necessary for reproducibility, it may be easier if the original authors publish the resources that they needed to run their experiment. Modern retrospective provenance systems [5, 22] do not yet provide a way of capturing or storing this information, although it would be straightforward to add. Knowing the total time the computational experiment is expected to take also helps future users know if they are getting "stuck" in a deadlock or infinite loop. We **do** report resource utilization requirements for the workflows in our dataset.

> We should use metadata to link the publication, funding, and authors to the workflow.

In the repositories that we studied, we could not find a machine-readable link between the workflow and the publication, funding, and authors. Linking the workflow to its publication and funding source would allow us to study the impact of policies on reproducibility. Git does store a history of the authors who touch the

[12] https://github.com/snakemake/snakemake/issues/44
[13] for example, https://github.com/marcodelapierre/toy-gpu-nf

code, but this would not include an advisor or other kinds of facilitators. Transitive Credit aims to solve this problem with JSON-LD [21], but it is not yet widely used.

6.1 Threats to Validity

The workflows we selected may not be representative of all workflows. We worked with two large registries and ran every workflow in each registry uniformly, but there may be a selectivity bias for workflows that end up being submitted to workflow registries. Still, our results are useful to points problems because problems for a community's best-kept workflows are likely problems for the other workflows as well.

We only test for reproducible crash-free execution. We cannot test research reproducibility because we do not have access to the original results, and we would need expertise to compare results from two runs to see if they really are equivalent. However, reproducible crash-free execution is a necessary condition for reproducible research results, and right now, only 51% of nf-core workflows and the 11% of SWC workflows have that.

The workflows we test may actually be reproducible, but our automated system could not figure out the right command to reproduce it. We suspect this case for many of instances where there is missing data. However, automatic reproducibility is still valuable because it is even easier to reproduce the experiment "out of the box," rather than having to dig around looking for data. Furthermore, automatic reproducibility is necessary for continuous integration (but, as noted earlier, continuous integration is not sufficient for reproducibility).

On the other hand, one might argue that in designing our automated system, we encoded too much information discovered from manually debugging failed workflows. For example, we found that SWC workflows often require Peppy and Pandas, just to subselect the data for input to the tasks. Since it is reasonable these packages might be installed on the user's machine, we added this into our software environment.

7 CONCLUSION

Reproducibility allows science to be self-correcting and helps us build on each other's results. While it intuitively seems that computational experiments should be perfectly reproducible, especially when compared to bench work, computational experiments are often the root of irreproducible research.

In this work, we investigate how reproducible workflows are in practice, by looking at workflows from two specific registries, nf-core and SWC. The fact that this experiment on reproducibility is possible is a testament to the improvements in tooling and community practices. The nf-core registry could be used as a good starting example of how communities standardize around common conventions and tooling. However, the current practice needs to be improved to get a higher degree of reproducibility. In particular, workflow authors should incorporate example data that runs "out of the box." In general, more work needs to be done on standardizing how to specify the means to reproduce a computational experiment.

REFERENCES

[1] J. Annan, D. Hargreaves, D. Lunt, A. Ridgwell, I. Rutt, and R. Sander. 2013. Editorial: The publication of geoscientific model developments v1.0. *Geoscientific Model Development* 6, 4 (Aug. 2013), 1233–1242. https://doi.org/10.5194/gmd-6-1233-2013 Publisher: Copernicus GmbH.

[2] STIG Authors. 2020. Red Hat Enterprise Linux 8 Security Technical Implementation Guide. https://www.stigviewer.com/stig/red_hat_enterprise_linux_8/2020-11-25/

[3] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. 2019. Parsl: Pervasive Parallel Programming in Python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/3307681.3325400

[4] Brett K. Beaulieu-Jones and Casey S. Greene. 2017. Reproducibility of computational workflows is automated using continuous analysis. *Nat Biotechnol* 35, 4 (April 2017), 342–346. https://doi.org/10.1038/nbt.3780 Number: 4 Publisher: Nature Publishing Group.

[5] Anila Sahar Butt and Peter Fitch. 2020. ProvONE+: A Provenance Model for Scientific Workflows. In *Web Information Systems Engineering – WISE 2020 (Lecture Notes in Computer Science)*, Zhisheng Huang, Wouter Beek, Hua Wang, Rui Zhou, and Yanchun Zhang (Eds.). Springer International Publishing, Cham, 431–444. https://doi.org/10.1007/978-3-030-62008-0_30

[6] CVE database. 2020. CVE - CVE-2020-14386. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-14386

[7] Ewa Deelman, Tom Peterka, Ilkay Altintas, Christopher D Carothers, Kerstin Kleese van Dam, Kenneth Moreland, Manish Parashar, Lavanya Ramakrishnan, Michela Taufer, and Jeffrey Vetter. 2018. The future of scientific workflows. *The International Journal of High Performance Computing Applications* 32, 1 (Jan. 2018), 159–175. https://doi.org/10.1177/1094342017704893 Publisher: SAGE Publications Ltd STM.

[8] Linux Developers. 2021. user_namespaces(7) - Linux manual page. https://www.man7.org/linux/man-pages/man7/user_namespaces.7.html

[9] Singularity Developers. 2023. Security in SingularityCE — SingularityCE Admin Guide 3.11 documentation. https://docs.sylabs.io/guides/latest/admin-guide/security.html

[10] Paolo Di Tommaso, Maria Chatzou, Evan W. Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. 2017. Nextflow enables reproducible computational workflows. *Nat Biotechnol* 35, 4 (April 2017), 316–319. https://doi.org/10.1038/nbt.3820 Number: 4 Publisher: Nature Publishing Group.

[11] Dave Dykstra. 2022. Apptainer Without Setuid. https://doi.org/10.48550/arXiv.2208.12106 arXiv:2208.12106 [cs].

[12] Philip A. Ewels, Alexander Peltzer, Sven Fillinger, Harshil Patel, Johannes Alneberg, Andreas Wilm, Maxime Ulysse Garcia, Paolo Di Tommaso, and Sven Nahnsen. 2020. The nf-core framework for community-curated bioinformatics pipelines. *Nat Biotechnol* 38, 3 (March 2020), 276–278. https://doi.org/10.1038/s41587-020-0439-x Number: 3 Publisher: Nature Publishing Group.

[13] Rafael Ferreira da Silva, Henri Casanova, Kyle Chard, Ilkay Altintas, Rosa M Badia, Bartosz Balis, Taina Coleman, Frederik Coppens, Frank Di Natale, Bjoern Enders, Thomas Fahringer, Rosa Filgueira, Grigori Fursin, Daniel Garijo, Carole Goble, Dorran Howell, Shantenu Jha, Daniel S. Katz, Daniel Laney, Ulf Leser, Maciej Malawski, Kshitij Mehta, Loic Pottier, Jonathan Ozik, J. Luc Peterson, Lavanya Ramakrishnan, Stian Soiland-Reyes, Douglas Thain, and Matthew Wolf. 2021. A Community Roadmap for Scientific Workflows Research and Development. In *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*. IEEE, St. Louis, MO, USA, 81–90. https://doi.org/10.1109/WORKS54523.2021.00016 arXiv:2110.02168 [cs] interest: 90.

[14] Rafael Ferreira da Silva, Kyle Chard, Henri Casanova, Daniel Laney, Dong H. Ahn, Shantenu Jha, William E. Allcock, Gregory Bauer, Dmitry Duplyakin, Bjoern Enders, Todd M. Heer, Eric Lancon, Sergiu Sanielevici, and Kevin Sayers. 2021. *Workflows Community Summit: Tightening the Integration between Computing Facilities and Scientific Workflows.* Technical Report ORNL/TM-2022/1832. Oak Ridge National Lab. (ORNL), Oak Ridge, TN (United States). https://doi.org/10.2172/1842590

[15] Rafael Ferreira da Silva, Loïc Pottier, Tainã Coleman, Ewa Deelman, and Henri Casanova. 2020. WorkflowHub: Community Framework for Enabling Scientific Workflow Research and Development. In *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*. IEEE, Georgia, USA, 49–56. https://doi.org/10.1109/WORKS51914.2020.00012

[16] Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. 2015. The Spack package manager: bringing order to HPC software chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2807591.2807623 interest: 80.

[17] José Manuel Gómez-Pérez, Esteban García-Cuesta, Aleix Garrido, José Enrique Ruiz, Jun Zhao, and Graham Klyne. 2013. When History Matters - Assessing Reliability for the Reuse of Scientific Workflows. In *The Semantic Web – ISWC 2013 (Lecture Notes in Computer Science)*, Harith Alani, Lalana Kagal, Achille Fokoue, Paul Groth, Chris Biemann, Josiane Xavier Parreira, Lora Aroyo, Natasha

Noy, Chris Welty, and Krzysztof Janowicz (Eds.). Springer, Berlin, Heidelberg, 81–97. https://doi.org/10.1007/978-3-642-41338-4_6

[18] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE '16)*. Association for Computing Machinery, New York, NY, USA, 426–437. https://doi.org/10.1145/2970276.2970358

[19] Konrad Hinsen. 2019. Dealing With Software Collapse. *Computing in Science & Engineering* 21, 3 (May 2019), 104–108. https://doi.org/10.1109/MCSE.2019.2900945 Conference Name: Computing in Science & Engineering.

[20] John P. Holden. 2013. Increasing Access to the Results of Federally Funded Scientific Research.

[21] Daniel S. Katz and Arfon M. Smith. 2015. Transitive Credit and JSON-LD. *Journal of Open Research Software* 3, 1 (Nov. 2015), e7. https://doi.org/10.5334/jors.by Number: 1 Publisher: Ubiquity Press.

[22] Farah Zaib Khan, Stian Soiland-Reyes, Richard O Sinnott, Andrew Lonie, Carole Goble, and Michael R Crusoe. 2019. Sharing interoperable workflow provenance: A review of best practices and their practical application in CWLProv. *GigaScience* 8, 11 (Nov. 2019), giz095. https://doi.org/10.1093/gigascience/giz095 interest: 98.

[23] M. S. Krafczyk, A. Shi, A. Bhaskar, D. Marinov, and V. Stodden. 2021. Learning from reproducing computational results: introducing three principles and the Reproduction Package. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 379, 2197 (March 2021), 20200069. https://doi.org/10.1098/rsta.2020.0069 Publisher: Royal Society.

[24] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. 2017. Singularity: Scientific containers for mobility of compute. *PLOS ONE* 12, 5 (May 2017), e0177459. https://doi.org/10.1371/journal.pone.0177459 Publisher: Public Library of Science.

[25] Johannes Köster and Sven Rahmann. 2012. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics* 28, 19 (Oct. 2012), 2520–2522. https://doi.org/10.1093/bioinformatics/bts480

[26] Haiyan Meng and Douglas Thain. 2017. Facilitating the Reproducibility of Scientific Workflows with Execution Environment Specifications. *Procedia Computer Science* 108 (Jan. 2017), 705–714. https://doi.org/10.1016/j.procs.2017.05.116

[27] Robert K. Merton. 1974. *The sociology of science: theoretical and empirical investigations* (4. dr. ed.). Univ. of Chicago Pr, Chicago.

[28] Alondra Nelson. 2022. Ensuring Free, Immediate, and Equitable Access to Federally Funded Research.

[29] Brian D. O'Connor, Denis Yuen, Vincent Chung, Andrew G. Duncan, Xiang Kun Liu, Janice Patricia, Benedict Paten, Lincoln Stein, and Vincent Ferretti. 2017. The Dockstore: enabling modular, community-focused sharing of Docker-based genomics tools and workflows. *F1000Res* 6 (Jan. 2017), 52. https://doi.org/10.12688/f1000research.10137.1

[30] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*. IEEE Press, Montreal, Quebec, Canada, 507–517. https://doi.org/10.1109/MSR.2019.00077 ISSN: 2574-3864.

[31] Line Pouchard, Sterling Baldwin, Todd Elsethagen, Shantenu Jha, Bibi Raju, Eric Stephan, Li Tang, and Kerstin Kleese Van Dam. 2019. Computational reproducibility of scientific workflows at extreme scales. *International Journal of High Performance Computing Applications* 33, 5 (April 2019), 763–776. https://doi.org/10.1177/1094342019839124 Institution: Brookhaven National Lab. (BNL), Upton, NY (United States) Number: BNL-211854-2019-JAAM Publisher: SAGE.

[32] Reid Priedhorsky and Tim Randles. 2017. Charliecloud: unprivileged containers for user-defined software stacks in HPC. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, Denver Colorado, 1–10. https://doi.org/10.1145/3126908.3126925

[33] Nicolas P. Rougier and Konrad Hinsen. 2019. ReScience C: A Journal for Reproducible Replications in Computational Science. In *Reproducible Research in Pattern Recognition (Lecture Notes in Computer Science)*, Bertrand Kerautret, Miguel Colom, Daniel Lopresti, Pascal Monasse, and Hugues Talbot (Eds.). Springer International Publishing, Cham, 150–156. https://doi.org/10.1007/978-3-030-23987-9_14

[34] ACM Inc. staff. 2020. Artifact Review and Badging. https://www.acm.org/publications/policies/artifact-review-and-badging-current

[35] Victoria Stodden and Sheila Miguez. 2014. Best Practices for Computational Science: Software Infrastructure and Environments for Reproducible and Extensible Research. *Journal of Open Research Software* 2, 1 (July 2014), e21. https://doi.org/10.5334/jors.ay Number: 1 Publisher: Ubiquity Press.

[36] Ana Trisovic, Matthew K. Lau, Thomas Pasquier, and Mercè Crosas. 2022. A large-scale study on research code quality and execution. *Sci Data* 9, 1 (Feb. 2022), 60. https://doi.org/10.1038/s41597-022-01143-6 Number: 1 Publisher: Nature Publishing Group.

[37] Jiawei Wang, Tzu-yang Kuo, Li Li, and Andreas Zeller. 2021. Assessing and restoring reproducibility of Jupyter notebooks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. Association for Computing Machinery, New York, NY, USA, 138–149. https://doi.org/10.1145/3324884.3416585

[38] Jun Zhao, Jose-Manuel Gomez-Perez, Khalid Belhajjame, Graham Klyne, Esteban Garcia-cuesta, Aleix Garrido, Kristina Hettne, Marco Roos, David De Roure, and Carole Goble. 2012. Why workflows break — understanding and combating decay in Taverna workflows. In *2012 IEEE 8th International Conference on E-Science (e-Science)*. IEEE, Chicago, IL, 9. https://doi.org/10.1109/eScience.2012.6404482

[39] Thomas H. Zurbuchen. 2022. SMD Policy Document SPD-41a.