How to enable inexpensive reproducibility for computational experiments

Samuel Grayson

2024 Oct 03

1 Problem

Reproducibility is essential to scientific experiments for three reasons:

1. The scientific community corrects false claims by applying scrutiny to judge past results. Scurtinizing an experiment often involves reproducing it, possibly with novel parameters. A reproducible experiments is easier to scrutinize, and therefore more likely to be correct, all else equal.
2. Science works by building off of the work of others. Extending one's work requires executing a modified version of their experiment. If reproducing the same experiment is difficult, one would expect executing an extended version to be even more so. Therefore, reproducible experiments may be easier to extend.
3. Scientific research aims to impact practice. Applying a novel technique to new data involves reproducing a part of the experiment that established the novel technique. Therefore, reproducible experiments may be easier to apply in practice.

Reproducibility also has costs, primarily in human labor needed to explain the experiment beyond that needed to disseminate the results. Working scientists balance the cost of reproducibility with the benefits to society or to themselves.

In real-world experiments, an multitude of possible factors must be controlled to find the desired result; it would be unfathomable that two instantiations of an experiment could give identical results. In contrast, for computational science experiments (from here on, **CS Exp**) on digital computers, while there are still many factors to control, perfect reproduction is quite fathomable. Despite this apparent advantage, CS Exps on digital computers still suffer low reproducibility rates.

The status quo will not change simply by arguing scientists should spend more time on reproducibility; those arguments are widely known are already taken into account. The status quo is a result of limited time and effort budgets. Nor do I have the power to change incentives in science funding policy unilaterally. However, by reducing the cost of reproducibility, scientists may achieve greater reproducibility with the same effort (fig. 1). **The goal of this work** is to improve the reproducibility of computational experiments achieved for an input of fixed effort.

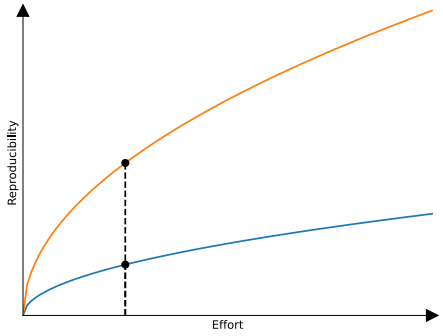

Figure 1: Effort-vs-reproducibility tradeoff curve in the status quo and with reproducibility tools. Introducing new technical solutions could shift the tradeoff from the blue curve to the orange curve.

The rest of the document will be structured as background (defining terms), prior approaches to decreasing the cost of reproducibility, my prior work on that problem, and proposed work.

2 Background

There have been conflicting sets of definitions for the term *reproducibility* (Plesser 2018). In this work, we use the Association for Computing Machinery (from here on, **ACM**) definition, if unspecified, although we discuss the other definitions in the prior work section. The ACM definition for **reproducibility** reads,

> 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.
>
> — (ACM Inc. staff 2020)

Experimental setup encompasses the aforementioned measurment procedure, measuring system, and operating conditions.

While reproducibility involves as a different team using the same experimental setup, there are similar terms to describe the other possible cases.

Term	Experimental method	Team
Repeatability	Same	Same
Reproducibility	Same	Different
Replicability	Different	Different

The ACM definition references more useful auxiliary terms, such as measurement procedure, operating conditions, and stated precision, so we use that for this work.

Naïvely, every CSE would be "reproducible", since there is some, possibly unknown, set of conditions under which another team can make the measurement; i.e., "make your environment bit-wise the same as ours". To prevent the definition from being vacuous, we only consider *explicitly stated* operating conditions.

These definitions derive from *metrology*, the science of measuring. I specialize some of the terms "measurement", "measurement procedure", and "operating conditions" in software context for computational science experiments:

- **Measurement procedure (for CSEs)**: The application code and user-interactions required to run the application code (if any).
- **Operating conditions (for CSEs)**: A set of conditions the computer system must implement to run a certain program. E.g., GCC 12 must be installed at `/usr/bin/gcc` compiled with certain features enabled.

  One may over-specify operating conditions without changing the reproducibility status. E.g., one might say their software requires GCC 12, but really GCC 12 or 13 would work. It can be difficult and often not necessary to know the necessary-and-sufficient set of operating conditions, so in practice, we usually have a set of conditions that is sufficient but not necessary to operate the experiment.

  Operating conditions can be eliminated by moving them to the measurement procedure. E.g., the program itself contains a copy of GCC 12. For the purposes of this work, the operating conditions are the "manual" steps that the user has to take to use the measurement procedure to make the measurement.
- **Measurement (for CSEs)**: Rather than offer a definition, we give some examples:
  - **Crash-freedom**: program produces result without crashing.
  - **Bit-wise equivalence**: output files and streams are bit-wise identical.
  - **Statistical equivalence**: overlapping confidence intervals, statistically consistent conclusions.
  - **Inferential equivalence**: whether the inference is supported by the output.
  - **Semantic equivalence**: equivalent according to the semantics of the defined data type (e.g., semantics of XML disregards whitespace in some contexts).

  In general, it is difficult to find a measurement that is both easy to assess and scientifically meaningful (tbl. 2).

Table 2: Measurments and their attributes.

Measurement	Easy to assess	Scientifically meaningful
Crash-freedom	Yes; does it crash?	Too lenient; could be no-crash but opposite result
Bit-wise equivalence	Yes	Too strict; could be off by one decimal point
Statistical equivalence	Maybe; need to know output format	Maybe; need to know which statistics *can* be off

Measurement	Easy to assess	Scientifically meaningful
Inferential equivalence	No; need domain experts to argue about it	Yes
Semantic equivalence	Depends on the semantics	Possibly

- **Composition of measurement procedures**: The outcome of one measurement may be the input to another measurement procedure.

  Composition happens in CSEs as well as in physical experiments. In physical experiments, one may use a device to calibrate (measure) another device, and use that other device to measure some scientific phenomenon. Likewise, In CSE, the output of compilation may be used as the input to another CSE. One can measure a number of relevant properties of the result of a software compilation. Composition of build processes motivates the following measurements

  - **Source equivalence**: compilation used the same set of source code as input.
  - **Behavioral equivalence**: the resulting binary has the same behavior as another one.
  - **Bit-wise equivalence**: As before, the binary is exactly the same as another one.

  E.g., suppose one runs `gcc main.c` on two systems and one system uses a different version of `unistd.h`, which is `#included` by `main.c`. The process (running `gcc main.c`) does not reproduce source-equivalent binaries, but it might reproduce behavior-equivalent binaries or bit-wise equivalent binaries (depending on how different `unistd.h`).

3 Related work

There is much related work that address reproducibility. I leverage the definitions and framework discussed above to contextualize each work. Still, there are significant gaps in prior work that this work exploits. Prior work can be divided into these categories, which we investigate in turn:

1. Characterizing reproducibility in theory or in practice
2. Studying approaches associated with proactively ensuring reproducibility

3.1 Characterizing reproducibility

Several prior works attempt to characterize reproducibility in theory or in empirical data.

3.1.1 Characterizing reproducibility in theory

Theoretical characterization begins by defining reproducibility.

Claerbout and Karrenbach give,

> Running the same software on the same input data and obtaining the same results
>
> — (Claerbout and Karrenbach 1992)

which many other works use. Replicability was defined later as

> Writing and then running new software based on the description of a computational model or method provided in the original publication, and obtaining results that are similar enough
>
> — (Rougier et al. 2017)

The ACM opted to use the terms as they were defined in metrology ("International Vocabulary of Metrology – Basic and General Concepts and Associated Terms (VIM)" 2012), which resulted in the same definitions being applied to opposite words as Claerbout and Karrenbach Plesser (2018). However, the ACM revised their definitions to be compatible with Claerbout and Karrenbach in 2020 (ACM Inc. staff 2020), so the definitions are mostly in consensus.

3.1.2 Empirical characterization of reproducibility

This group of related work seeks to characterize the degree of reproducibility or a proxy for reproducibility in a sample of CSEs empirically (tbl. 3). However, the type of reproducibility assessed varies widely between studies. A proxy variable could be "whether the source is available", since this is a necessary but not sufficient condition for reproducibility.

Table 3: Prior works characterizing empirical reproducibility.

Publication	N	Subjects	Population	Repro measurement assessed or proxy variable	Level
Vandewalle, Kovacevic, and Vetterli (2009)	134	Article artifacts	2004 ed. of img. proc. journal	Code availability	9%
Zhao et al. (2012)	92	Taverna workflows	myExp. 2007 – 2012	Crash-free execution	29%
Collberg and Proebsting (2016)	508[1]	Source code	2012 eds. of CS journals	Crash-free compilation	48%
Gundersen and Kjensmo (2018)	400	Article artifacts	2013 – 2016 AI journals	Sufficient description	24%[2]

[1] I consider the total sample to be only those papers whose results were backed by code, did not require special hardware, and were not excluded due to overlapping author lists, since those are the only ones Collberg and Proebsting attempted to reproduce. I am considering OK<30 and OK>30 as "reproducible crash-free building" because codes labelled OK>Author were not actually reproduced on a new system (Collberg and Proebsting 2016).

[2] This figure is the average normalized "reproducibility score", based on whether the method, data, and experiment, were available. If it were 1, all the papers in the sample would be have method, data, and experiment availability, and if it were 0, none would be.

Publication	N	Subjects	Population	Repro measurement assessed or proxy variable	Level
Pimentel et al. (2019)	863,878[3]	Jupyter notebooks	GitHub	Same-stdout execution	4%
Krafczyk et al. (2021)	5	Articles artifacts	Journal of Comp. Phys.	Figures and tables semantic equivalence	70%[4]
Wang et al. (2021)	3,740[5]	Jupyter notebooks	GitHub	Crash-free execution	19%
Trisovic et al. (2022)	2,109[6]	R scripts	Harvard Dataverse	Crash-free execution	12%

Note that crash-freedom is prevalent because it is the easiest to automatically assess. Source-availability requires human-intervention to test, so studies on crash-freedom simply begin from a corpus for which source code is available. Note that stdout-equivalence may be feasible because Jupyter Notebooks bundle the stdout with the code; otherwise, the stdout is usually thrown away.

Overall, these studies show that reproducibility is a significant problem.

Some of the studies, like Zhao et al., investigate the *reason* why some samples were not reproducible (Zhao et al. 2012), finding

Reason	Proportion of failures
Unavailable/inaccessible 3rd party resource	41%
Updated 3rd party resource	7%
Missing example data	14%
Insufficient execution environment	12%
Insufficient metadata	27%

Zhao et al. (2012) remark that provenance could preserve or enable repair for several classes of failures, including unavailability of 3rd party resources and insufficient descriptions. I investigate this approach in my proposed work.

[3] I consider the total sample to be only notebooks that were valid, pure Python, and had an unambiguous order, since those are the ones Pimentel et al. attempt to reproduce (Pimentel et al. 2019.)

[4] This figure is the average number of computational elements (figures and tables) that wereeproduced to semantic equivalence in each paper.

[5] I consider the total sample to be only notebooks that had dependency information and used Python 3.5 or later, since those are the only ones Wang et al. attempted to reproduce (Wang et al. 2021).

[6] I consider the total samples to be the set of *un-repaired* codes, since those are the ones that actually exist publicly. Also, we *include* codes for which the time limit was exceeded; reproduction was attempted and not successf in those conditions.

3.2 Proactively ensuring reproducibility

There are several proposed approach associated with proactively facilitating reproducibility:

Approach	Aspect of reproducibility addressed
Scientific clouds	Reduces non-portable operating condions
Source artifact archival	Ensure attainability of operating condition
Workflow managers	Simplify measurement procedure
Provenance	Identifies operating conditions
Record/replay executions	Reduces operating conditions
Virtualization/containerization	Reduces non-portable operating conditions
Digital notebooks	Explicates measurements

3.3 Scientific clouds

Scientific clouds facilitate reproducibility by replacing a complex set of operating conditions ("install this, configure that") with a single operating condition: log in to a cloud system. The cloud system hosts a controlled environment with the rest of the operating conditions already met.

However, storing the environments and running executions in them is expensive. Either the users have to pay, in which case the computational environment is not "freely accessible", or some institution may sponsor public-access. Institutional grants for public-access may be indefinite or it may only last for a certain amount of time due to funding constraints (tbl. 6).

Table 6: Various scientific clouds found in prior work, by creation date.

Scientific Cloud	URL	Lifetime on Archive.org	Main publication
Binder	mybinder.org	2018 – present	Jupyter et al. (2018)
Chameleon Cloud (federated)	chameleoncloud.org	2015 –resent	Keahey et al. (2020)
PhenoMeNal portal	phenomenal-h2020.eu	2017 – 2023	Peters et al. (2019)
WholeTale	WholeTale.org	2016 – present	Brinckman et al. (2019)
GridSpace2	gs2.plgrid.pl	2015 – 2020	Ciepiela, Zaraska, and Sulka (2012)
Collage Authoring Environment	collage.elsevier.com	2013 – 2014	Nowakowski et al. (2011)
RunMyCode	RunMyCode.org	2012 – 2024	Stodden, Hurlin, and Perignon (2012)
SHARE		2011 – 2023	Van Gorp and Mazanek (2011)

Scientific Cloud	URL	Lifetime on Archive.org	Main publication
Galaxy (federated)	usegalaxy.org	2007 – present	The Galaxy Community et al. (2024)
GenePattern	GenePattern.org	2006 – present	Reich et al. (2006)

An improvement on this model is the *federated model* (as in The Galaxy Community et al. (2024)), where the compute infrastructure could be provided by multiple parties which themselves may be free-access, pay-for-access, or locally hosted servers. So long as the user can reserve compute resources on any *one* of them, they can import and run the CSE. Federation doesn't alleviate the funding problem, but it does spread out the funding problem; if the original grant runs out, some other institution can pick up the torch. If all else fails, technically savvy individuals can run servers for just themselves (so-called "self-hosting"). However, federation trades off with standardization and consistency; a CSE may only work with certain infrastructure providers.

3.3.1 Literate programming and digital notebooks

The idea of literate programming according to Knuth is to mix narrative and explanation into code.

> Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.
>
> — (Knuth 1984)

Digital notebooks or electronic lab notebooks extend this to also include multimedia, such as graphs, widgets, and other data visualizations. Popular digital notebooks include: Jupyter, Amazon Sagemaker, Google Colab, Deepnote, Hex, Databricks Notebooks, DataCamp Workspace, JupyterLab, HyperQuery, Jetbrains Datalore, kaggle, NextJournal, Noteable, nteract, Observable, Query.me, VS Code notebooks, Mode Notebooks, Querybook, Zeppelin, Count, Husprey, Pluto.jl, Polynote, Zepl[7].

Digital notebooks facilitate reproducibility by:

- encoding the measurement itself, which may otherwise be printed to `stdout` and not distributed.
- encoding the explanation for the measurement procedure, making it easier to adapt if errors or difference arise.
- automating data analysis steps that would otherwise by typed into an interpreter and not saved.

However, digital notebooks still require a separate way of specifying the software environment (operating conditions). Repo2docker, famously used by the scientific cloud Binder, standardizes a way of specifying the software environment for notebooks contained in a repository by leveraging off-the-shelf package managers (Jupyter et al. 2018). I discuss

[7] List at https://datasciencenotebook.org/

package managers as a solution foreproducibility later on; the point still stands that literate programming requires other methods to manage the software environment in order to attain reproducibility.

3.3.2 Source code archive

Source code archives (tbl. 7) facilitate reproducibility by making operating conditions namable and satisfiable. Namability approaches include Digital Object Identifiers (DOI), Archival Resource Key (ARK), Uniform Resource Names (URNs), and persistent URLs (PURLs).

Archiving old versions of core software helps conditions remain satisfiable, since old versions may otherwise get deleted or become unfindable.

Table 7: Source Archives by creation date.

Source Code archive	Description	Lifetime	Main publication
Software Heritage Archive	Collects codes in other sources	2016 – present	(Di Cosmo et al. 2020)
Zenodo	User-uploaded research codes	2013 – present	(Sicilia, García-Barriocanal, and Sánchez-Alonso 2017)
FigShare	User-uploaded research codes	2012 – present	(Singh 2011)

Storing source code in central archives shares some of the disavdantages of scientific clouds, but to a lesser extent. Merely storing scientific software is much easier than offering to execute it on-demand. Indeed, all of the source archiving solutions we found in prior literature are still alive today; the same cannot be said for scientific clouds.

While source code archival does help with unfulfillable conditions, it still requires someone or something to actually set up, configure, build, and install the source code to actually satisfy the condition "install this version of that software". Other methods should work in concert, falling back on source code archival if the code is no longer available from the original hoster.

3.3.3 Workflow managers

A workflow manager is a system that executes a directed graph of loosely coupled, coarse nodes. Usually, each node is a process and each edge is a file.

Workflow managers facilitate reproducibility by:

- Explicitly stating parts of the measurement procedure that would otherwise be unwritten (which script to run, in what order)
- Letting systems reason about the inputs and outputs of the measurement procedure

Mölder et al. (Mölder et al. 2021) classify workflow engines by their interface (tbl. 8) which we extend. The interface can be graphical, a library in a general purpose language (GPL), a domain specific language (DSL), or a data definition language (such as YAML). I reclassified

the workflow managers in Mölder's system-independent class, since that does not refer to the interface, and we added other common workflow managers.

Table 8: Common (more than 1 example in any registry on https://workflows.communities/registries) workflow managers classified by interface. DSL stands for "Domain-Specific Language", and GPL stands for "General-Purpose Language". More workflow managers available here: https://github.com/meirwah/awesome-workflow-engines

Workflow Manager	Interface
Galaxy	Graphical
KNIME	Graphical
Scipion	Graphical
COMPS (in Java)	Library in GPL
Parsl (in Python)	Library in GPL
Dask (in Python)	Library in GPL
Pegasus (in Python, Java, R)	Library in GPL
Luigi (in Python)	Library in GPL
Nextflow	DSL
Snakemake	DSL
Workflow Description Language (WDL)	DSL
Bpipe	DSL
Common Workflow Language (CWL) (in YAML)	Data Definition Language

Additionally, there are other ways to classify workflows, including parallelism (task, data, both, neither), static vs dynamic scheduling (Liu et al. 2015). Workflows can be subjectively evaluated based on clarity, well-formedness, predictability, recordability, reportability, reusability, scientific data modelling, and automatic optimization (McPhillips et al. 2009).

Workflow managers often represent a departure from what the user is currently doing, and therefore take significant effort to adopt. The dependency problem may be worsened because workflow managers introduce another dependency to be managed.

3.3.4 Provenance

Computational provenance (henceforth, **provenance**) of a computational artifact is the processes that generated the computational artifact, the inputs to that provenance, and the provenance of those inputs (recursively). See fig. 2 for example. Provenance facilitates reproducibility by explicitly stating the inputs (operating conditions).

There are four levels at which one can capture computational provenance (Freire et al. 2008) (tbl. 9):

- **application-level**: modifying an application to report provenance data.
- **workflow-level** or **language-level**: leveraging a workflow engine or programming language to report provenance data.
- **system-level**: leveraging an operating system to report provenance data.
- **service-level**: recording provenance at the boundaries between services in a system.

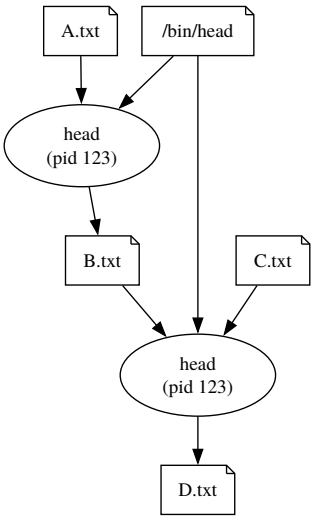

Figure 2: Example of a provenance graph for `D.txt`. It includes the process that generates `D.txt` (in this case `head`), the inputs to that process (in this case `B.txt` and `C.txt`), and the provenance of `B.txt` and `C.txt`.

Table 9: Provenance capturing systems for CSE in prior work.

Provenance system	Level
REDUX	Workflow-level
Swift	Workflow-level
VisTrails	Workflow-level
Kepler	Workflow-level
Taverna	Workflow-level
PASS	System-level
ES3	System-level
Sumatra	System-level
PASOA	Service-level
Karma	Workflow-level or service-level
noWorkflow	Language-level

More provenance systems and classifications are discussed in the First Provenance Challenge (Moreau et al. 2008).

Most of these provenance systems are implemented at a workflow-level or service-level, so they are only applicable to users of that workflow or service. System-level could be more generally usable by any program running in that system. However, neither Sumatra, PASS, nor ES3 leverage the provenance to create automatic replay. These methods help reproducibility, but they do not deliver "push button" reproducibility, which we need to extract the maximum value from the minimum amount of work.

noWorkflow attempts to capture provenance for arbitrary Python programs (Murta et al.

2015). Python provenance could be useful for projects whose data processing is primarily Python, however it would not be able to track data past that boundary. Many computational experiments involve dataflow between multiple processes (e.g., a shell script).

3.3.5 Record/replay tools

Record/replay is a feature where the user executes an unmodified program in a "record" environment, which creates a reproducibility package. The reproducibility package can be stored or distributed, and later on, the user can "replay" the same execution from it.

Record/replay	Capture method
CDE	Ptrace
RR	Ptrace
CARE	Ptrace
Sciunit	Ptrace
ReproZip/PTU	Ptrace
OPUS	Library interposition
PROV-IO	Library interposition
bpftrace	eBPF
PASS	Kern. mod.
CamFlow	Kern. mod.

While record/replay is a convenient user-interface, none of the implementations are fast and run without super user access. Scientific users rarely have super user privileges on shared hardware; root "in container" is not strong enough to install kernel modules or eBPF (extended Berkely Packet Filter) programs. The overhead of conventional record/replay tools is often more than 2x, making them infeasible to use in many cases.

One might argue that record/replay "allows" users to be messy and merely preserves the mess perfectly (Douglas Thain 2015). Simply copying the entire filesystem, for example, may work (the operating conditions are likely contained in the filesystem), but an entire filesystem is inconvenient to transfer, compose, and execute. An ideal record/replay tool should allow users to introspect and simplify their computational environment.

3.3.6 Virtualization and containerization

System virtualization facilitates reproducibility by encapsulating an entire system (and thus its operating conditions) in a distributable sandbox. System virtualization drivers include QEMU, VirtualBox, VMWare Fusion, Parallels for Mac, etc.

Containerization is similar, but it only encapsulates Linux userspace, reusing the hosts Linux kernel. On non-Linux hosts, the container is nested inside a virtualized Linux instance. Containerization has weaker isolation than system virtualization but better performance and lower storage. Container engines include Docker, Podman, and Singularity.

In either case, either the system image (VM image, Docker image, etc.) or instructions to build the system image (e.g., `Vagrantfile`, `Dockerfile`, `Singularityfile`, etc.) has to be distributed.

- Distributing the image is difficult because it is large, expensive to store, and expensive to transfer. DockerHub, a popular storage space for Docker images, recently changed their policy to remove images that had not been pulled or pushed in the past six months (de Morlhon 2020a), although enforcement was delayed until 2021 (de Morlhon 2020b).
- The build instructions, although easy to distribute, require *another* reproducibility strategy to be reproducible. It simply redirects the problem of CSE irreproducibility to Docker image irreproducibility.

4 My completed work

In this section, I outline my completed work on this subject (fig. 3).

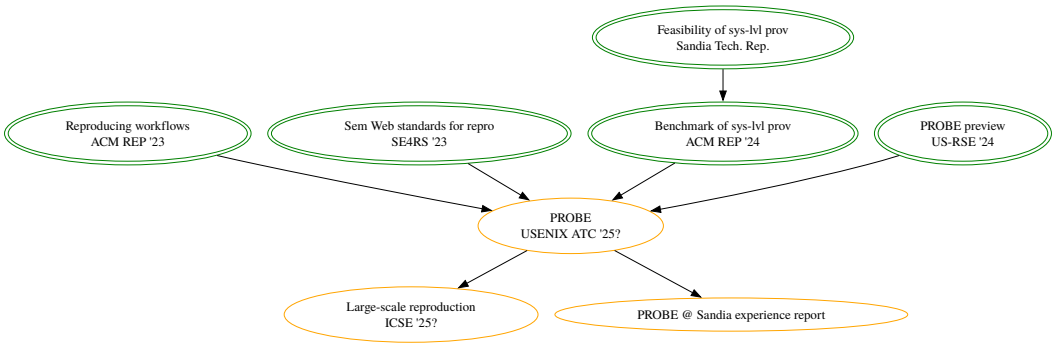

Figure 3: My prior and proposed work. The prior work is green and double-circled; the proposed work is orange.

4.1 Examining automatic reproductions of workflows

This section summarizes "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 the 2023 ACM Conference on Reproducibility and Replicability (ACM REP '23). Association for Computing Machinery, New York, NY, USA, 74–84. https://doi.org/10.1145/3589806.3600037"

We attempted to automatically run run a large corpus of workflows. We used two workflow registries listed on https://workflows.community/, in particular: Snakemake Workflow Catalog (SWC) and nf-core (Grayson et al. 2023). Of these, about half had at least one non-crashing revision (tbl. 11). For those whose most recent revision did not crash, we looked how far back we could go before we found a crashing revision. Evaluating old software today approximates the failure rate of evaluating today's software in the future. We found that workflows had a median failure rate of a couple of years (fig. 4).

Table 11: Statistics on workflows in SWC and nf-core.

Quantity	All	SWC	nf-core
# workflows	101	53	48
# revisions	584	333	251
% of revisions with no crash	28%	11%	51%
% of workflows with at least one non-crashing revision	53%	23%	88%

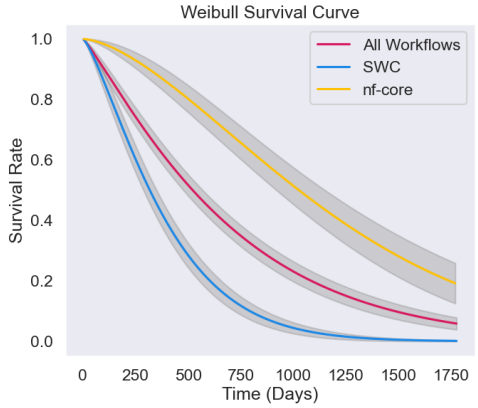

Figure 4: Survival curve analysis of workflows

When the workflows failed, we classified the crash reason (tbl. 12). The most common reason is simply a missing input. The workflow manager could not help in this case, because the missing input is not an internal node, but a top-level node, where the workflow manager would read a file from the outside system. Provenance tracing could help automatically upload inputs that a workflow needs to run.

Table 12: Workflow failure reasons

Kind of crash	Proportion
Missing input	32%
Conda environment unsolvable	11%
Unclassified reason	8%
Timeout reached	7%
Container error	6%
Other (workflow script)	6%
Other (containerized task)	1%
Network resource changed	1%
Missing dependency	0.5%
No crash	28%
Total	100%

Conclusion: we learn that while workflows, containers, and package managers are useful, errors can still persist in their use. For example, the workflows may refer to Singularity images that no longer exist. We learn that missing input data is a the most common error cause, so any approach attempting to improve automatic reproducibility should address that.

4.2 Review of system-level provenance tracers tools

This section summarizes "Samuel Grayson, Faustino Aguilar, Reed Milewicz, Daniel S. Katz, and Darko Marinov. 2024. *A benchmark suite and performance analysis of user-space provenance collectors*. In Proceedings of the 2nd ACM Conference on Reproducibility and Replicability (ACM REP '24). Association for Computing Machinery, New York, NY, USA, 85–95. https://doi.org/10.1145/3641525.3663627"

We executed a rapid literature review to systematically search for system-level provenance collectors in prior literature (Grayson et al. 2024). We arrived at a list of 45, and then:

- Selected only those that worked for Linux (39).
- From those, selected those which did not use VMs (37).
- From those, selected those which did not require source-code recompilation (35).
- From those, selected those which did not require special hardware (31).
- From those, selected those which had source code available (20).
- From those, selected those which did not require a custom kernel (15).
- From those, attempted to reproduce 10.
- From those, selected those which successfully completed all of the benchmarks (5): ReproZip, strace, fsatrace, CARE, and RR-debugger.

Note that RR-debugger, strace, fsatrace, and CARE are not true system-level provenance collectors, because they do not export a provenance graph. RR-debugger and CARE allow automatic replay of the recording, while strace and fsatrace do not; the latter two are strictly "informational". In the interest of having a more complete benchmark set, we cast a wide net and include these provenance-adjacent projects.

Unfortunately, there is no standard set of benchmarks used to evaluate system-level provenance tracers. Therefore, we also extracted all of the benchmarks used in any work returned by this search (tbl. 13).

Table 13: Benchmarks from prior works and this work.

Prior works	This work	Category
12	yes	HTTP server/traffic
10	yes	HTTP server/client
10	yes	Compile user packages
9	yes	I/O microbenchmarks (lmbench + Postmark)
9	no	Browsers
6	yes	FTP client
5	yes	FTP server/traffic
5	yes	Un/archive
5	yes	BLAST
5	yes	CPU benchmarks
5	yes	Coreutils and system utils

Prior works	This work	Category
3	yes	cp
2	yes	VCS checkouts
2	no	Sendmail
2	no	Machine learning workflows (CleanML, Spark, ImageML)
1	no	Data processing workflows (VIC, FIE)

Between these five, we ran the extracted benchmarks in the provenance tracers (tbl. 14). We measured the percent overhead from native case, so a value of 50% means the new runtime would be the native runtime times 1.5. We can see that none of the existing provenance collectors are fast enough for practical use, except for fsatrace.

Table 14: Performance of different provenance collectors.

Benchmark	Native	fsatrace	CARE	strace	RR	ReproZip
BLAST	0	0	2	2	93	8
Tar Unarchive	0	4	44	114	195	149
Python import	0	5	85	84	150	346
VCS checkout	0	5	71	160	177	428
Compile w/Spack	0	-1	119	111	562	359
Postmark	0	2	231	650	259	1733
cp	0	37	641	380	232	5791
Others not shown	...	...	...	...	...	...
Geometric mean	0	0	45	66	46	193

The salient difference is that the other provenance tracers use ptrace to capture the underlying calls, while fsatrace uses `LD_PRELOAD`. ptrace involves two context switches on every system call, because the tracee and tracer occupy two separate processes fig. 5. `LD_PRELOAD` involves *no* extraneous context switches, because the tracee and tracer occupy the same process fig. 6. Therefore, we build on the underlying technology of fsatrace in our future work.

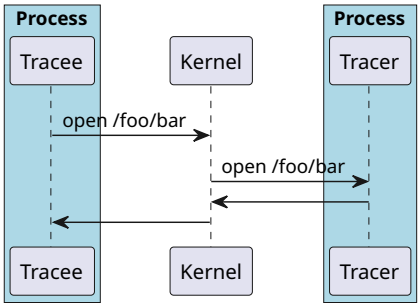

Figure 5: Tracing an `open` syscall using `ptrace`

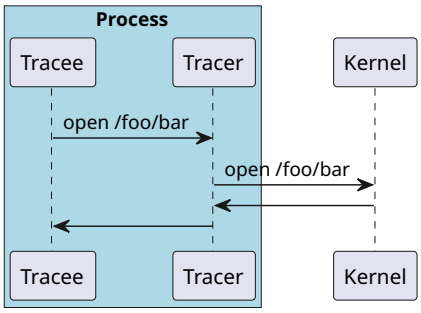

Figure 6: Tracing an `open` syscall using library interpositioning

Conclusion: System-level provenance tracers in prior work either require super-user access or are two slow for practical use. However, library interpositioning is a promising direction to avoid both issues.

5 Proposed work

See the single-circled nodes in fig. 3.

After having gathered empirical evidence on why CSEs fail to be reproducible and gathered evidence on state-of-the-art provenance tracers, it is time to set off on future work.

- 2025 Dec 01: stablize PROBE's core funcitonality
- 2025 Dec 20: stabalize reproduced C&P dataset
- 2025 Jan 14: USENIX ATC '25 submission deadline (PROBE)
- 2025 Mar 22 (expected): ICSE '25 submission deadline (reproduce C&P)
- 2025 May 28 (expected): eScience '25 submission deadline (PROBE @ Sandia)

5.1 Provenance & Replay Observation Engine (PROBE)

I am working on a novel, system-level provenance tracer that uses `LD_PRELOAD` called PROBE. The tracer is capable of tracing simple applications and extracting a provenance dataflow graph. I need to work on:

- tracing more complex applications
- tracing remote applications (by wrapping remote accesses)
- replaying the trace on other machines
- converting the trace to a workflow

I plan to leverage provenance standards such as RO-crates and W3C PROV.

I would evaluate PROBE based on its performance and the set of configuration variables that it isolates in the replay environment. Both of which will be compared relative to other record/replay tools

Eventually, I want to put PROBE to use in practice at Sandia National Labs, perhaps writing an experience report about translating research into practice.

5.2 Reproducibility in computer systems research

Collberg and Proebsting attempt a large-scale reproduction of the crash-free build-environments used in artifacts from publications in top CS journals and conferences (Collberg and Proebsting 2016). They find that source availability is the dominant issue preventing reproducibility, and labor required to package a prominent reason for source non-availability.

Has the landscape changed since then? What is the decay rate of the artifacts gathered by Collberg and Proebsting? Like the (Grayson et al. 2023), one could estimate the decay rate from two years-apart samples.

Of particular interest to me, how well would PROBE be able to snapshot and reproduce the executions? We sought to interpose all relevant library calls we could find, but perhaps we missed some. This could be evaluated by repeating the build from the PROBE recording on a new machine without internet access.

5.3 PROBE @ Sandia: Experience report

An experience report of actually using PROBE at Sandia National Labs would bolster my research portfolio by showing that PROBE can actually work in practice. The claims of automatic containerized replay is somewhat advanced, so some reviewers may be reluctant to believe it until they see it.

Additionally, this application would demonstrate my ability to bring research ideas over the "valley of death" into practice. Research is valuable as research, but it can be more valuable if it impacts practice in some way. I want translational computer science (translating ideas from theory to practice) to be a major theme in my future career.

Bibliography

ACM Inc. staff. 2020. "Artifact Review and Badging." August 24, 2020. https://www.acm.org/publications/policies/artifact-review-and-badging-current.

Brinckman, Adam, Kyle Chard, Niall Gaffney, Mihael Hategan, Matthew B. Jones, Kacper Kowalik, Sivakumar Kulasekaran, et al. 2019. "Computing Environments for Reproducibility: Capturing the 'Whole Tale'." *Future Generation Computer Systems* 94 (May): 854–67. https://doi.org/10.1016/j.future.2017.12.029.

Ciepiela, Eryk, Leszek Zaraska, and Grzegorz D. Sulka. 2012. "GridSpace2 Virtual Laboratory Case Study: Implementation of Algorithms for Quantitative Analysis of Grain Morphology in Self-Assembled Hexagonal Lattices According to the Hillebrand Method." In *Building a National Distributed e-Infrastructure–PL-Grid*, edited by Marian Bubak, Tomasz Szepieniec, and Kazimierz Wiatr, 7136:240–51. Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-28267-6_19.

Claerbout, Jon F., and Martin Karrenbach. 1992. "Electronic Documents Give Reproducible Research a New Meaning." In *SEG Technical Program Expanded Abstracts 1992*, 601–4. Society of Exploration Geophysicists. https://doi.org/10.1190/1.1822162.

Collberg, Christian, and Todd A. Proebsting. 2016. "Repeatability in Computer Systems Research." *Communications of the ACM* 59 (3): 62–69. https://doi.org/10.1145/2812803.

Di Cosmo, Roberto, Morane Gruenpeter, Bruno P Marmol, Alain Monteil, Laurent Romary, and Jozefina Sadowska. 2020. "Curated Archiving of Research Software Artifacts :

Lessons Learned from the French Open Archive (HAL)." In *IDCC 2020 - International Digital Curation Conference*. Dublin, Ireland. https://doi.org/10.2218/ijdc.v15i1.698.

Douglas Thain, Peter Ivie. 2015. "Techniques for Preserving Scientific Software Executions: Preserve the Mess or Encourage Cleanliness?" University of Notre Dame. https://doi.org/10.7274/R0CZ353M.

Freire, Juliana, David Koop, Emanuele Santos, and Cláudio T. Silva. 2008. "Provenance for Computational Tasks: A Survey." *Computing in Science & Engineering* 10 (3): 11–21. https://doi.org/10.1109/MCSE.2008.79.

Grayson, Samuel, Faustino Aguilar, Reed Milewicz, Daniel S. Katz, and Darko Marinov. 2024. "A Benchmark Suite and Performance Analysis of User-Space Provenance Collectors." In *Proceedings of the 2nd ACM Conference on Reproducibility and Replicability*, 85–95. Rennes France: ACM. https://doi.org/10.1145/3641525.3663627.

Grayson, Samuel, 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 the 2023 ACM Conference on Reproducibility and Replicability*, 74–84. ACM REP '23. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3589806.3600037.

Gundersen, Odd Erik, and Sigbjørn Kjensmo. 2018. "State of the Art: Reproducibility in Artificial Intelligence." *Proceedings of the AAAI Conference on Artificial Intelligence* 32 (1). https://doi.org/10.1609/aaai.v32i1.11503.

Heroux, Michael, Lorena Barba, Manish Parashar, Victoria Stodden, and Michela Taufer. 2018. "Toward a Compatible Reproducibility Taxonomy for Computational and Computing Sciences." SAND–2018-11186, 1481626, 669580. https://doi.org/10.2172/1481626.

"International Vocabulary of Metrology – Basic and General Concepts and Associated Terms (VIM)." 2012. JCGM. https://doi.org/10.59161/JCGM200-2012.

Jupyter, Project, Matthias Bussonnier, Jessica Forde, Jeremy Freeman, Brian Granger, Tim Head, Chris Holdgraf, et al. 2018. "Binder 2.0 - Reproducible, Interactive, Sharable Environments for Science at Scale." In, 113–20. Austin, Texas. https://doi.org/10.25080/Majora-4af1f417-011.

Keahey, Kate, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, et al. 2020. "Lessons Learned from the Chameleon Testbed." In *2020 USENIX Anuual Technical Conference*. USENIX. https://www.usenix.org/conference/atc20/presentation/keahey.

Knuth, D. E. 1984. "Literate Programming." *The Computer Journal* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.

Krafczyk, M. S., 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): 20200069. https://doi.org/10.1098/rsta.2020.0069.

Liu, Ji, Esther Pacitti, Patrick Valduriez, and Marta Mattoso. 2015. "A Survey of Data-Intensive Scientific Workflow Management." *Journal of Grid Computing* 13 (4): 457–93. https://doi.org/10.1007/s10723-015-9329-8.

McPhillips, Timothy, Shawn Bowers, Daniel Zinn, and Bertram Ludäscher. 2009. "Scientific Workflow Design for Mere Mortals." *Future Generation Computer Systems* 25 (5): 541–51. https://doi.org/10.1016/j.future.2008.06.013.

Mölder, Felix, Kim Philipp Jablonski, Brice Letcher, Michael B. Hall, Christopher H. Tomkins-Tinch, Vanessa Sochat, Jan Forster, et al. 2021. "Sustainable Data Analysis with Snakemake." F1000Research. https://doi.org/10.12688/f1000research.29032.2.

Moreau, Luc, Bertram Ludäscher, Ilkay Altintas, Roger S. Barga, Shawn Bowers, Steven Callahan, George Chin, et al. 2008. "Special Issue: The First Provenance Challenge." *Concurrency and Computation: Practice and Experience* 20 (5): 409–18. https://doi.org/10.1002/cpe.1233.

Morlhon, Jean-Laurent de. 2020a. "Scaling Docker's Business to Serve Millions More Developers: Storage." *Docker Blog* (blog). August 24, 2020. https://www.docker.com/blog/scaling-dockers-business-to-serve-millions-more-developers-storage/.

———. 2020b. "Docker Hub Image Retention Policy Delayed, Subscription Updates." *Docker Blog* (blog). October 22, 2020. https://www.docker.com/blog/docker-hub-image-retention-policy-delayed-and-subscription-updates/.

Murta, Leonardo, Vanessa Braganholo, Fernando Chirigati, David Koop, and Juliana Freire. 2015. "noWorkflow: Capturing and Analyzing Provenance of Scripts." In *Provenance and Annotation of Data and Processes*, edited by Bertram Ludäscher and Beth Plale, 71–83. Lecture Notes in Computer Science. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-16462-5_6.

Nowakowski, Piotr, Eryk Ciepiela, Daniel Harężlak, Joanna Kocot, Marek Kasztelnik, Tomasz Bartyński, Jan Meizner, Grzegorz Dyk, and Maciej Malawski. 2011. "The Collage Authoring Environment." *Procedia Computer Science* 4: 608–17. https://doi.org/10.1016/j.procs.2011.04.064.

Peters, Kristian, James Bradbury, Sven Bergmann, Marco Capuccini, Marta Cascante, Pedro de Atauri, Timothy M D Ebbels, et al. 2019. "PhenoMeNal: Processing and Analysis of Metabolomics Data in the Cloud." *GigaScience* 8 (2). https://doi.org/10.1093/gigascience/giy149.

Pimentel, João Felipe, 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*, 507–17. MSR '19. Montreal, Quebec, Canada: IEEE Press. https://doi.org/10.1109/MSR.2019.00077.

Plesser, Hans E. 2018. "Reproducibility Vs. Replicability: A Brief History of a Confused Terminology." *Frontiers in Neuroinformatics* 11. https://www.frontiersin.org/articles/10.3389/fninf.2017.00076.

Reich, Michael, Ted Liefeld, Joshua Gould, Jim Lerner, Pablo Tamayo, and Jill P Mesirov. 2006. "GenePattern 2.0." *Nature Genetics* 38 (5): 500–501. https://doi.org/10.1038/ng0506-500.

Rougier, Nicolas P., Konrad Hinsen, Frédéric Alexandre, Thomas Arildsen, Lorena Barba, Fabien C. Y. Benureau, C. Titus Brown, et al. 2017. "Sustainable Computational Science: The ReScience Initiative." *PeerJ Computer Science* 3 (December): e142. https://doi.org/10.7717/peerj-cs.142.

Sicilia, Miguel-Angel, Elena García-Barriocanal, and Salvador Sánchez-Alonso. 2017. "Community Curation in Open Dataset Repositories: Insights from Zenodo." *Procedia Computer Science* 106: 54–60. https://doi.org/10.1016/j.procs.2017.03.009.

Singh, Jatinder. 2011. "FigShare." *Journal of Pharmacology and Pharmacotherapeutics* 2 (2): 138–39. https://doi.org/10.4103/0976-500X.81919.

Stodden, Victoria, Christophe Hurlin, and Christophe Perignon. 2012. "RunMyCode.org: A Novel Dissemination and Collaboration Platform for Executing Published Computational Results." In *2012 IEEE 8th International Conference on E-Science*, 1–8. Chicago, IL, USA: IEEE. https://doi.org/10.1109/eScience.2012.6404455.

The Galaxy Community, Linelle Ann L Abueg, Enis Afgan, Olivier Allart, Ahmed H Awan,

Wendi A Bacon, Dannon Baker, et al. 2024. "The Galaxy Platform for Accessible, Reproducible, and Collaborative Data Analyses: 2024 Update." *Nucleic Acids Research* 52 (W1): W83–94. https://doi.org/10.1093/nar/gkae410.

Trisovic, Ana, Matthew K. Lau, Thomas Pasquier, and Mercè Crosas. 2022. "A Large-Scale Study on Research Code Quality and Execution." *Scientific Data* 9 (1): 60. https://doi.org/10.1038/s41597-022-01143-6.

Van Gorp, Pieter, and Steffen Mazanek. 2011. "SHARE: A Web Portal for Creating and Sharing Executable Research Papers." *Procedia Computer Science* 4: 589–97. https://doi.org/10.1016/j.procs.2011.04.062.

Vandewalle, Patrick, Jelena Kovacevic, and Martin Vetterli. 2009. "Reproducible Research in Signal Processing." *IEEE Signal Processing Magazine* 26 (3): 37–47. https://doi.org/10.1109/MSP.2009.932122.

Wang, Jiawei, 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*, 138–49. ASE '20. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3324884.3416585.

Zhao, Jun, 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)*, 9. Chicago, IL: IEEE. https://doi.org/10.1109/eScience.2012.6404482.