Predictive continuous testing to mitigate software collapse in scientific software

Samuel Grayson
Univerity of Illinois Urbana Champaign
Urbana, IL
0000-0001-5411-356X

Daniel S. Katz
University of Illinois Urbana Champaign
Urbana, IL
0000-0001-5934-7525

Reed Milewicz
Sandia National Laboratories
Albuquerque, NM
rmilewi@sandia.gov

Darko Marinov
Univerity of Illinois Urbana Champaign
Urbana, IL
0000-0001-5023-3492

***Abstract*—Software tends to break or "collapse" over time, even if it is unchanged, due to non-obvious changes in the computational environment. Collapse in computational experiments undermines long-term credibility and hinders day-to-day operations. We propose to create the first public dataset of automatically executable scientific experiments. We explain how that data can be used to identify best practices, make continuous testing feasible, and repair broken programs, in order to increase the reproducibility of computational expeirments.**

***Index Terms*—reproducibility; software reliability; continuous testing**

I. Introduction

Software tends to break over time, even if it is unchanged, due to non-obvious changes in the computational environment. This phenomenon is called "software collapse" [1], because software with an unstable foundation is analogous to a building with unstable foundation. Software collapse is not a significant problem in some domains; it is acceptable if Google returns slightly different results one day to the next. But in the scientific domain, software collapse could manifest as irreproducible results or unreliable software[1], which not only undermine long-term credibility of science but also hinder its day-to-day operations.

1. **Undermines long-term credibility**: More than half of scientists surveyed across all fields develop software for their research [4]. [2] If computational experiments are allowed to collapse, scientists cannot independently verify or build on each others' results. This undermines two fundamental norms of science identified by Merton, organized skepticism and communalism [5], that make science self-correcting. In recent years, this has manifested itself as the ongoing reproducibility crisis[3] in computational science [6], which damages the long-term credibility of science [7].
2. **Hinders day-to-day operations**: Consider scientists tasked with securing their nations' nuclear stockpile. They might create a simulation that tests if a physical part is going to properly preform a critical function for nuclear storage. The physical part might last several decades, but the software often collapses much faster than that. As our understanding of material science improves, they might want to reassess if the simulation still predicts the part preforms its function properly given our improved understanding. If the simulation experienced software collapse, this will likely need to be fixed, despite the software not changing. Fixing the software may be difficult or impossible, especially if the original developer is retired.

Unfortunately, software collapse appears to be widespread in the computational science domain. Zhao et al. studied software collapse computational of experiments deposited in the myExperiment registry [8]. They found that 80% of the experiments in their selection did not work, for a variety of causes: change of third-party resources, unavailable example data, insufficient execution environment, and insufficient metadata; of these, change of third-party resources caused the most failures, such as when a step in an experiment referenced data from another server through the internet which was no longer available.

The problem of irreproducibility in scientific computing is not solely technical: the cultural norms around preserving

[1] In this article, we use Claerbout's terminology to define ir/reproducibility [2]: one can use the same code in a different computational environment to get the same result. Reproducibility is called "replicability" by some authors; see Plesser [3] for a discussion of terminology. Un/reliable, on the other hand, just refers to whether the software can fail to produce a result.

[2] DSK: 90+% of researchers use research software, 50% develop it, research papers are filled with software mentions, research funders spend a significant fraction of their budget on software...

[3] Contrary to the name, Irreproducible and unreliable contribute to the so-called "reproducibility crisis" in science.

scientific software and attitudes of funding agencies play significant roles in the decision to invest in software sustainability and reproducibility. Our work examines technical solutions which should be part of a holistic effort to address policy, economic, and social factors that drive software collapse in science. Such a solution could be proactive or reactive: a *proactive solution* would control and preserve the environment or application in order to to ensure reproducibility as software ages, whereas a *reactive solution* would wait until reproducibility fails and try to fix that or alert human developers. The following are examples of state-of-the-art proactive tools:

- **Snapshotting the environment**: Container images (e.g., Docker), VM images, CDE [9], and Sumatra [10] attempt to snapshot the entire computational environment. Then, one can ship the entire filesystem to another user so they can reproduce the execution. However, this approach is heavyweight with filesystem snapshots as large as 50 Gb, as it needs to record a large chunk of the filesystem. Finally, these are difficult to modify and audit.
- **Specifying construction of the environment**: `Dockerfile`s and install scripts let the user specify instructions to construct the computational environment enclosing software. However, these instructions are UNIX commands, which can be non-deterministic themselves[4], e.g. `pip install`.
- **Functional package managers:** Functional package managers (e.g., Nix, Guix, Spack) is a restricted form of environment construction which only permits certain UNIX capabilities. For example, Nix only lets users download from the internet if they provide a hash of the expected outcome; if this hash is different, Nix errors because this execution is not going to reproduce th previous one. While this is useful for setting up the environment, this approach is only applied to the installation phase not the actual execution phase. It would be too burdensome to provide hash for every network access within the application, if the application talks to a database for example. [5] [6]

The most straightforward way to improve reproducibility is through proactive solutions[7], but none of these solutions can mitigate non-determinism due to network resources, pseudorandomness, and parallel program order. Zhao et al. showed that first of these, networked resources, is the most common cause of software collapse as well [8], so irreproducibility due to the network cannot be ignored. Henkel et al. find 25% of Dockerfiles in their already limited sample still fail to build [11]. Therefore, important computational experiments should be protected from collapse by proactive *and* reactive solutions.

Continuous testing is a reactive solution to software collapse that is robust to networked resources, pseudorandomness, and parallel program order. [8] One could imagine running the computational experiment periodically to assess if the experiment is both not crashing and still producing the same results. The major drawback is increased computational cost. However, one can always lower the frequency of testing, which trades off computational resources with efficacy of finding bugs. Additionally, one could test mission-critical experiments more frequently than other experiments. If one could predict which workflows were more likely to break, one could also prioritize testing on that basis.

[SAG: Explain difference with traditional CI.]

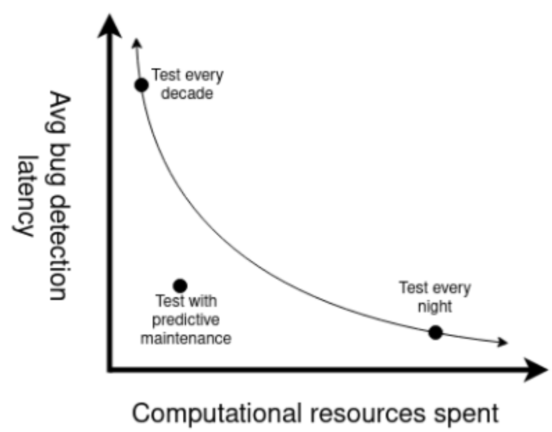

Figure 1: Predicting the rate of software collapse can reduce the resource utilization and increase efficacy of continuous testing.

Hinsen suggests that most code should build on reliable, well-tested libraries can provide some degree of resistance to collapse [1]. In practice, many experiments fall into collapse despite their best effort to build on reliable foundations. If that level of reliability is insufficient, one can add continuous testing to help get more reliability. [9]

[4] Although many people believe Docker gives them reproducibility [11], Docker itself never claims that `Dockerfile`s are reproducible; The term "reproducible" and "reproducibility" only occur three times in Docker's documentation at the time of this writing, and none of them are referring to reproducing the same result from running a `Dockerfile` twice. One occurrence references to ability to reproduce an environment on another machine by pulling the same container image, not by running the `Dockerfile` twice. Distributing the container image is described in the previous bullet (snapshotting the environment).

[5] DSK: maybe talk about version pinning?

[6] SAG: Refer to the literature on reproducing Jupyter/IPython notebooks

[7] SAG: citations

[8] DSK: I'm not sure about the parallel program order part. SAG: this can be done by injecting randomness into the program schedule; recent work does something like that. I have yet to go back and find that paper, and weave it into the narrative in this paragraph.

[9] SAG: Explain why we need dataset (to improve reproducibility through continuous testing, automatic program repair, and identify best practices).

This paper will build a dataset of software collapse of computational experiments and answer the following research questions:

- **RQ1 measure rate of software collapse:** What are typical rates of software collapse over time? This number is not well-known, since the last experiment to measure it was Zhao et al., and we have new reproducibility technology (NextFlow over Taverna).
- **RQ2 predict rate of software collapse:** Can we predict the rate of decay for a project based on its history (if available) and code? A predictive model is important for the next research question. The model should function on a "cold start", where we know nothing about the computational experiments historical results, but it should be able to learn from those historical runs if they are present.
- **RQ3 optimize continuous testing:** Can we improve the efficiency of continuous testing by predicting the rate of decay? This could be useful for instutions, such as national labs, wanting to ensure their computational experiments remain valid while using resources efficiently.
- **RQ4 identify best practices:** What are the best practices that improve reproducibility? This lets us make recommendations that are empirically backed.
- **RQ5 attempt automatic repair:** In what fraction of the cases does automatic repair work? Automatic repair could let one run old workflows off-the-shelf with no modification.

II. Methods (Collecting data)

We plan to collect data on software collapse of computational experiments by automatically running computational experiments from public registries. These registries include:

- nf-core: *TODO: describe each of these (one sentence).*
- Dockstore
- Snakemake Catalog
- WorkflowHub
- myExperiment
- PegasusHub
- Sandia's internal repository
- Globus Flows
  - https://www.globus.org/platform/services/flows
  - https://anl-braid.github.io/braid/

We cannot take one computational experiment and simulate it one, five, and ten years into the future. Instead, we will look for historical revisions[10] of an experiment from one, five, or ten years ago and simulate it today. All of the registries above store historical revisions[11] of the workflow. We make a *time symmetry* assumption: historical rates of change will be similar to the future rate of change. It is likely that some will still work and some sill fail, due to software collapse.

We will run the following pseudo-code to collect the data. Then we we will analyze it as described in the next section. Finally, we plan to publish the raw data we collect for other researchers.

```
for registry in registries:
    for experiment in registry:
        for revision in experiment:
            for i in range(num_repetitions):
                execution = execute(revision)
                data.append((
                    execution.date,    execution.output,
                    execution.logs,    execuiton.resource_utilization,
                    revision.date,     revision.code,
                    experiment.name,   registry.name,
                ))
```

III. Analysis

- **RQ1 measure rate of collapse:** We plan to replicate the quantities described by Zhao et al. [8] to see if these have changed since that work, or if they are different for workflows[12]: proportion of broken experiments, and proportion of breakages due to each reason (volatile third-party resources, missing example data, missing execution environment, insufficient description). To this, we add "reproducible results" as a new "level" of success, beyond merely not crashing. We also plan to extend the failure classification of Zhao et al. by going into deeper subcategories. We will also study how the proportion of broken experiments changes with time.
- **RQ2 predict rate of collapse:** We will develop predictive models based on the history of failures, staleness, properties of the code in the revision, and other determinants to predict the probability that a given experiment will fail. We will use information theory criteria to quantify the difference from our predicted distribution to the actual distribution.
- **RQ3 improve continuous testing:** We can improve resource utilization of continuous testing by using our dataset to predict the rate of collapse of various computational experiments. Testing experiments prone to failure more often than reliable ones could save computational resources while maintaining approximately the same degree of reliability in all experiments.
- **RQ4 identify best practices:** We can also use this data to identify practices that improve the reproducibility and longevity of computational experiments. We will use a "Bayes net" to test for confounding causal variables. [13]
- **RQ5 attempt automatic repair:** Once we know what kinds of failure are possible, we can also inves-

[10]DSK: versions?
[11]DSK: versions?

[12]SAG: Let's simplify this
[13]SAG: what best practices? Try looking at workflow manager, cyclomatic complexity, SLoC, grammatical size, reproducibility tools (docker, requirements.txt with pinned packages, singularity)

tigate automatic repair. Our dataset will contain the output logs for each failure. Therefore, we can apply similar techniques to Shipwright [11], such as using a language model to categorize many failures into a few clusters.

A. *Threats to Validity*

There are a number of threats to the validity of our work and planned results.

1. Our time symmetry assumption may not hold. With contemporary efforts on reproducibility, future rates of change may be markedly less than past rates of change. While our computed rates of change will be underestimates, those underestimates can still be useful as bounds. Our method will also be useful, unchanged, for future studies.
2. It is possible that our sample is not representative of the real world of computational experiments. However, we are casting the widest net we can by systematically pulling many experiments from several registries. Still, there is a selection bias in which workflows end up in registries. The model has some factors based on the population and some based on the actual history of the experiment. Its initial guess when there is no history would be biased by our selection, but in the long run it would learn the characteristics of the actual experiment.

IV. Conclusion

Software collapse is an important yet understudied problem. We don't know the rate of software collapse in contemporary computational experiments. In order to do any research in this area, we need to build standard, communal datasets on software collapse.

The dataset would indicate how various reactive solutions compare, allowing us to identify the best practices that correlate with reproducibility. However, no proactive solution is perfect, so we also look at the reactive solution of continuous testing. The dataset would also allow us to optimize continuous testing such that it is feasible. Finally, when the continuous testing finds a failure, the automatic repair we plan to prototype would help fix that failure.

A. *Future Work*

[SAG: TODO]

- Use the dataset for workflows that work, with timeout and resource utilization
- Automatic scale-down experiments
- Automatically tune error thresholds
- Autotuning experiments

V. References

[1] K. Hinsen, "Dealing With Software Collapse," *Computing in Science & Engineering*, vol. 21, no. 3, pp. 104–108, May 2019, doi: 10.1109/MCSE.2019.2900945.

[2] J. F. Claerbout and M. Karrenbach, "Electronic documents give reproducible research a new meaning," in *SEG Technical Program Expanded Abstracts 1992*, Jan. 1992, pp. 601–604, doi: 10.1190/1.1822162 [Online]. Available: http://library.seg.org/doi/abs/10.1190/1.1822162. [Accessed: Jun. 01, 2022]

[3] H. E. Plesser, "Reproducibility vs. Replicability: A Brief History of a Confused Terminology," *Frontiers in Neuroinformatics*, vol. 11, 2018 [Online]. Available: https://www.frontiersin.org/articles/10.3389/fninf.2017.00076. [Accessed: Oct. 11, 2022]

[4] S. Hettrick, "Software-saved/Software_in_research_survey_2014: Software In Research Survey." Zenodo, Feb. 2018 [Online]. Available: https://zenodo.org/record/1183562. [Accessed: May 26, 2022]

[5] R. K. Merton, *The sociology of science: Theoretical and empirical investigations*, 4. Dr. Chicago: Univ. of Chicago Pr, 1974.

[6] C. Collberg and T. A. Proebsting, "Repeatability in computer systems research," *Communications of the ACM*, vol. 59, no. 3, pp. 62–69, Feb. 2016, doi: 10.1145/2812803. [Online]. Available: https://dl.acm.org/doi/10.1145/2812803. [Accessed: May 27, 2022]

[7] S. Ritchie, *Science Fictions: How Fraud, Bias, Negligence, and Hype Undermine the Search for Truth*, Illustrated edition. New York: Metropolitan Books, 2020.

[8] J. Zhao *et al.*, "Why workflows break — understanding and combating decay in Taverna workflows," in *2012 IEEE 8th International Conference on E-Science (e-Science)*, Oct. 2012, p. 9, doi: 10.1109/eScience.2012.6404482 [Online]. Available: https://www.research.manchester.ac.uk/portal/en/publications/why-workflows-break--understanding-and-combating-decay-in-taverna-workflows(cba81ca4-e92c-408e-8442-383d1f15fcdf)/export.html

[9] P. Guo and D. Engler, "CDE: Using System Call Interposition to Automatically Create Portable Software Packages," in *2011 USENIX Annual Technical Conference*, Jun. 2011 [Online]. Available: https://www.usenix.org/legacy/events/atc11/tech/final_files/GuoEngler.pdf

[10] A. Davison, "Automated Capture of Experiment Context for Easier Reproducibility in Computational Research," *Computing in Science & Engineering*, vol. 14, no. 4, pp. 48–56, Jul. 2012, doi: 10.1109/MCSE.2012.41. [Online]. Available: http://ieeexplore.ieee.org/document/6180156/. [Accessed: Jul. 08, 2022]

[11] J. Henkel, D. Silva, L. Teixeira, M. d'Amorim, and T. Reps, "Shipwright: A Human-in-the-Loop System for Dockerfile Repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, May 2021, pp. 1148–1160, doi: 10.1109/ICSE43902.2021.00106.