

A reactive approach to identifying and mitigating software collapse computational science

Samuel Grayson, Department of Computer Science, Univerity of Illinois Urbana Champaign
Daniel S. Katz, NCSA & CS & ECE & iSchool, Univerity of Illinois Urbana Champaign [d.katz@uiuc.edu](#)
Darko Marniov, Department of Computer Science, Univerity of Illinois Urbana Champaign [dmarniov@uiuc.edu](#)
Reed Milewicz, Sandia National Laboratories rmilewi@sandia.gov

Abstract

I will write this last. TODO

Introduction

More than half of scientists surveyed across all fields develop software for their research [1]. Unfortunately, the code they develop tends to break over time, even if it is unchanged, due to non-obvious changes in the computational environment.

¹ This phenomenon is called “software collapse” [2], because software with an unstable foundation is analogous to a building with unstable foundation. ² This breakage could manifest as irreproducible³ results. ⁴

If computational experiments are allowed to collapse, scientists cannot independently verify or build on those results. Thus, software collapse undermines two fundamental norms of science identified by Merton, organized skepticism and communalism [4]. Software collapse is a technical factor that contributes to the ongoing reproducibility crisis in computational science [5], which hinders the credibility of science [6]. ⁵

Zhao et al. studied software collapse computational of experiments deposited in the myExperiment registry [7]. They found that 80% of the experiments in their selection did not work, for a variety of causes: change of third-party resources,

¹DSK: this seems like it is saying that this is problem for scientists, while it’s really an issue for all software developers.

²DSK: and all scientific software has an unstable foundation, unless it’s monolithic.

³In this article, we use Claerbout’s terminology [3]. “Reproducibility” means anyone can use the same code to get the same result. [DSK: this defintion is a bit loose - Also, I like the terminology discussion in <https://www.frontiersin.org/articles/10.3389/fninf.2017.00076/full>]

⁴DSK: It’s more likely to manifest as code that doesn’t build or run - though I agree it could also lead to different results

⁵DSK: I might prefer to cite <https://www.amazon.com/Science-Fictions-Negligence-Undermine-Search/dp/1250222699> as more broad and a bit less political.

unavailable example data, insufficient execution environment, and insufficient metadata. Of these, change of third-party resources caused the most failures. This included a step in the experiment that referenced data from another server through the internet which was no longer available.

Part of the problem is technical: people who want their software to be reliable do not necessarily know how to achieve that, given their resource constraints. We suggest a technical solution, which should be a part of a holistic solution. The technical solution could be proactive or reactive: a proactive solution would change something about the environment or application to provide determinism, whereas a reactive solution would seek to detect non-determinism and alert human developers. Proactive solutions are preferable, but to date, no proactive solution will eliminate a large case of irreproducibility. Therefore, we need both proactive and reactive solutions.

Proactive solutions

- **Docker:** A Dockerfile is a set of UNIX commands and auxiliary commands that specify how to build a Docker image. However, these instructions are UNIX commands, which can be non-deterministic themselves⁶, e.g. `pip install`. Docker cannot 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 [7]. This is empirically validated, as Henkel et al. find 25% of Dockerfiles in their already limited sample still fail to build [8].⁷
- **Filesystem snapshot:** Container images (e.g., Docker, Singularity), functional package managers (e.g., Nix, Guix), CDE [9], Sumatra [10], and chroot containers spawn a program in a view of the filesystem that they control. 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. Furthermore, this does not control network resources, pseudorandomness, and parallel program order. Finally, these are difficult to work with to

Reactive solution

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

⁶Docker itself never claims that `Dockerfiles` are reproducible; it only says “reproducible” three times in [their documentation](#) at the time of writing, and none of them are referring to reproducing the same result from running a `Dockerfile` twice.

⁷DSK: maybe talk about version pinning?

⁸DSK: I’m not sure about the parallel program order part.

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.

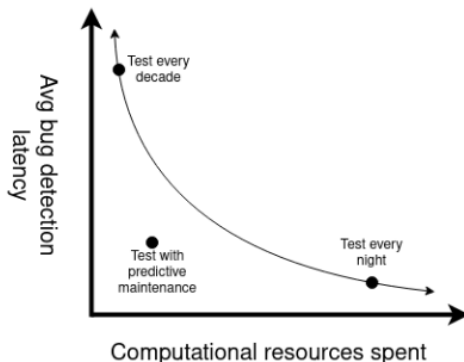


Figure 1: Predicting the rate of software collapse can reduce the resource utilization and increase efficacy of continuous testing.

Methods

We want to⁹ 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 revisions¹⁰ of an experiment from one, five, or ten years ago and simulate it today. All of the registries above store historical revisions¹¹ of the workflow. We make a *time*

⁹DSK: We plan to? We have begun to? Our aim is to?

¹⁰DSK: versions?

¹¹DSK: versions?

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 will fail, due to software collapse.

We will run the following pseudo-code to collect the data. Then 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,
                    execution.resource_utilization,
                    revision.date,
                    revision.code,
                    experiment.name,
                    experiment.interpreter,
                    registry.name,
                ))
```

Analysis

We plan to replicate the quantities described by Zhao et al. [7] to see if these have changed since that work, or if they are different for workflows¹²: 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.

We can improve resource utilization of continuous testing by using our dataset to predict the rate of collapse of various computational experiments. We will develop predictive models based on the staleness, properties of the code in the revision, and other determinants to predict the probability that a given experiment will fail. 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.

Hinsen suggests that most code should build on reliable, well-tested libraries can provide some degree of resistance to collapse [2]. In practice, many experiments

¹²Let’s simplify this

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.

Once we know what kinds of failure are possible, we can also investigate automatic repair. Our dataset will contain the output logs for each failure. Therefore, we can apply similar techniques to Shipwright [8], such as using a language model to categorize a large number of failures into a small number of clusters.

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.

Threats to Validity

There are a number of threats to the validity of our work and planned results. First, our time symmetry assumption may not hold. With all of the contemporary focus and effort around 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.

Second, 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 a large number experiment 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.

TODO: other threats to validity.

References

- [1] S. Hettrick, “Softwaresaved/Software_in_research_survey_2014: Software In Research Survey.” Zenodo, Feb. 2018 [Online]. Available: <https://zenodo.org/record/1183562>. [Accessed: May 26, 2022]
- [2] K. Hinsien, “Dealing With Software Collapse,” *Computing in Science & Engineering*, vol. 21, no. 3, pp. 104–108, May 2019, doi: [10.1109/MCSE.2019.2900945](https://doi.org/10.1109/MCSE.2019.2900945).
- [3] 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](https://doi.org/10.1190/1.1822162) [Online]. Available: <http://library.seg.org/doi/abs/10.1190/1.1822162>. [Accessed: Jun. 01, 2022]

- [4] R. K. Merton, *The sociology of science: Theoretical and empirical investigations*, 4. Dr. Chicago: Univ. of Chicago Pr, 1974.
- [5] 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](https://doi.org/10.1145/2812803). [Online]. Available: <https://dl.acm.org/doi/10.1145/2812803>. [Accessed: May 27, 2022]
- [6] J. P. A. Ioannidis, “Why Most Published Research Findings Are False,” *PLOS Medicine*, vol. 2, no. 8, p. e124, Aug. 2005, doi: [10.1371/journal.pmed.0020124](https://doi.org/10.1371/journal.pmed.0020124). [Online]. Available: <https://journals.plos.org/plosmedicine/article?id=10.1371/journal.pmed.0020124>. [Accessed: Sep. 14, 2022]
- [7] 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](https://doi.org/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](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)
- [8] 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](https://doi.org/10.1109/ICSE43902.2021.00106).
- [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](https://doi.org/10.1109/MCSE.2012.41). [Online]. Available: <http://ieeexplore.ieee.org/document/6180156/>. [Accessed: Jul. 08, 2022]