Continuous testing to mitigate software collapse in scientific software

***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. These techniques increase the replicability of computational experiments.**

***Index Terms*—reproducibility; software reliability; 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 an unstable foundation. In the scientific domain, software collapse could manifest as an non-repeatable (and thusly irreplicable) experiment[1], which not only undermine long-term credibility of science but also hinder its day-to-day operations.

Undermines long-term credibility: Computational experiments are widely used in many scientific disciplines. More than 90% of scientists surveyed across all fields use research software and 50% develop software for their research experiments [3]. If computational experiments are allowed to collapse, scientists cannot independently verify or build on each other's results, which requires replicability. This undermines two fundamental norms of science identified by Merton, organized skepticism and communalism [4], that make science self-correcting. In recent years, this has manifested itself as the ongoing reproducibility crisis[2] in computational science [5], which damages the long-term credibility of science [6].

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 perform 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 performs its function properly given our improved understanding. If the simulation experienced software collapse, this will 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 is widespread in the computational science domain. 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, 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.

A. Prior state-of-the-art

The problem of non-replicability in scientific computing is not solely technical: the cultural norms around preserving scientific software and attitudes of funding agencies play significant roles in the decision to invest in software sustainability and replicability.

Our work examines technical techniques which should be part of a holistic effort to address policy, economic, and social factors that drive software collapse in science. These techniques solution can be proactive or reactive: a *proactive technique* would control and preserve the environment or application to ensure repeatability as software ages, whereas a *reactive technique* would wait until repeatability fails and try to fix that. Note that repeatability is a

[1]In this article, we use ACM's terminology [2]: **Repeatable:** one can execute the computational experiment again in the same computational environment to get an approximately equivalent result. **Replicability:** one can execute the computational experiment in a different computational environment to get approximately equivalent results. **Reproducibility:** one can execute a novel computational experiment to come to the same conclusion Reproducibility implies replicability, which implies repeatability, which implies that the software does not crash. While the converses are not true, repeatability is a necessary step towards repeatability and repeatability towards reproducibility, so achieving repeatability should make it easier to achieve reproducibility.

[2]Despite these definitions, the "reproducibility crisis'' in computational science is primarily due to non-replicable computational experiments. If they were at least replicable but not necessarily reproducible, independent researchers chould scrutinize or build on those results.

necessary condition and a step towards replicability, but it is not replicability. The following are examples of state-of-the-art proactive tools:

Snapshotting the environment: Container images (e.g., Docker), virtual machine images, CDE [8], and Sumatra [9] attempt to snapshot the entire computational environment. Then, one can ship the entire filesystem to another user so they can repeat 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.

Specify environment in scripts: `Dockerfile`s and shell 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[3], e.g., `pip install`. Henkel et al. find 25% of Dockerfiles in their already limited sample still fail to build [10].

Specify environment in package managers: Package managers such as Pip, Conda, Nix, Guix, Spack, etc. allow users to specify the computational environment to run their experiment, like a restricted form of scripting. However, the most common of these (Pip, Conda, and Nix) allow users to specify packages *without* pinning a specific version and require extra steps (often not taken!) to lock the versions. Even if the versions are uniquely pinned, data is often not distributed as a package but pulled from ephemeral resources on the internet,

The most straightforward way to improve replicability is through proactive techniques [8], [9], but none of these techniques can mitigate non-determinism due to network resources, pseudorandomness, and parallel program order. Zhao et al. showed that the first of these, networked resources, is the most common cause of software collapse as well [7], so non-replicability due to the network cannot be ignored. Therefore, important computational experiments should be protected from collapse by proactive *and* reactive techniques. Here are reactive techniques:

Continuous testing: Automated systems can run the computational experiment continuously to assess if the experiment is both not crashing and still producing the same results (repeatability). Continuous testing is robust to more sources of non-determinism, including networked resources, pseudorandomness, and parallel program order.[4] Testing is usually a part of continuous integration/continuous deployment (CI/CD), but the continuous testing we are proposing here differs from CI/CD because our proposed continuous testing is triggered periodically, while the CI/CD is triggered when the code is changed. CI/CD mitigates software regressions, which are due to *internal changes*, but continuous testing mitigates software collapse, which is due to *external changes*. The major drawback is increased computational cost, since running a computational experiment can be expensive. However, if one could predict which workflows were more likely to break, one could also prioritize testing on that basis, an optimization we term *predictive continuous testing*.

[3]Distributing the container image is described in the previous bullet.

[4]Non-determinism in the pseudorandom number generator and program schedule can be injected by the environment.

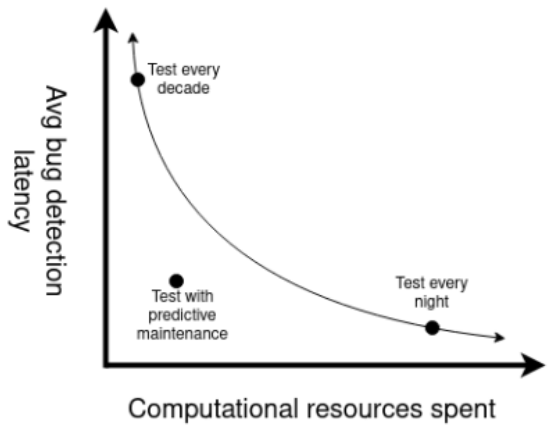

Figure 1: Predicting the rate of software collapse can reduce resource utilization and increase efficacy of continuous testing.

Automatic program repair: Automatic program repair seeks to encode solutions for common sources of errors. This has been done successfully in other domains [10]

Hinsen suggests that building 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.

B. Research Questions

We would like to study the usage and efficacy of these techniques and even improve them. However, there is a dearth of data on the repeatability of computational experiments. There are experimental registries, but they do not store prior results, so we cannot tell if the experiment is repeatable. This paper will build a dataset of software collapse of computational experiments, which will allow us to answer the following research questions:

RQ1: 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 replicability techniques (NextFlow builds on Taverna).

RQ2: When software collapses, what is the immediate technical cause? Zhao et al. studies these at a high-level, and we plan to replicate those categories as well as delve into more subcategories. For example, when a third-party resource is unavailable, we will assess whether that resource is *data* or a *software dependency*.

RQ3 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 operate from a "cold start," where we know nothing about the computational experiment's historical results, but also be able to learn from historical executions if they are present.

RQ4: Can we improve the efficiency of continuous testing by predicting the rate of decay? This could be useful for intuitions, such as national labs, wanting to ensure their computational experiments remain valid while using resources efficiently.

RQ5: What are the best practices that improve replicability? This lets us make recommendations that are empirically backed.

RQ6: 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. Method

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

- nf-core
- Dockstore
- Snakemake Catalog
- WorkflowHub
- myExperiment
- PegasusHub
- Globus Flows
- The internal repository of private laboratories[5]

We cannot take one computational experiment and simulate it one, five, and ten years into the future. Instead, we will look for historical versions of an experiment from one, five, or ten years ago and simulate it today. The registries above store historical versions of the workflow. We make a *time symmetry* assumption: historical rates of change will be like the future rate of change. 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 version in experiment:
            for i in range(num_repetitions):
                execution = execute(version)
                data.append((
                    execution.date,   execution.output,
                    execution.logs,   execuiton.res_usage,
                    version.date,     version.code,
                    experiment.name,  registry.name,
                ))
```

III. Analysis

RQ1: We plan to replicate the experiment described by Zhao et al. [7], which assesses if the computational experiments are replicable in our environment. To this, we add "repeatable results" as a new column. We will also study how the proportion of broken experiments changes with time. Note that a failure could indicate collapse, or it could indicate that the experiment never worked in the first place, due to incomplete metadata. We can model this using a Bayesian framework that permits either possibility (never working or collapse) as an unobserved random variable.

RQ2: We will examine some of the non-replicable experiments and classify their causes. While we would like to classify all the non-replicable experiments, this may not be practical. Instead, we will analyze a random sample.

RQ3: We will develop predictive models based on the history of failures, staleness, properties of the code in the version, 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.

RQ4: We can improve resource utilization of continuous testing by using our dataset to predict the rate of collapse of various computational experiments. Since we would have data on the computational cost (runtime and RAM) of each experiment, we can analytically simulate "what if we test X every Y days." Then we can simulate a system that tests each computational experiment in a frequency based on its failure rate and computational cost.

RQ5: We can also use this data to identify practices that improve the replicability of computational experiments. We plan to examine choice of workflow manager, cyclomatic complexity, significant lines of code, choice of replicability tools (docker, `requirements.txt` with pinned packages, singularity), and other factors.

RQ6: Once we know what kinds of failures 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 [10], such as using a language model to categorize many failures into a few clusters.

A. Threats to Validity

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

[5]These experiments can be included in our aggregated analysis, but not in the raw dataset.

1. Our time symmetry assumption may not hold. With contemporary efforts on replicability, 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 it would eventually learn the characteristics of the actual experiment.

IV. Conclusion

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

The dataset would indicate how various reactive techniques compare, allowing us to identify the best practices that correlate with replicability. However, no proactive technique is perfect, so we also look at reactive techniques such as continuous testing and automatic program repair. 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

There is a plethora of exciting future work to be done on this topic and with this dataset. Many studies of replicability such as Collberg and Proebsting [5], Zhao et al. [7], Henkel et al. [10] require a dataset of computational experiments with metadata that supports automatic execution. However, each had to construct their own datasets. This dataset mined from 6 community standard experimental registries could serve as a starting point for future research on replicability. Notably, our dataset will contain a distribution of the CPU time, RAM, and disk space needed to run the experiment, so researchers can make informed requests to batch schedulers (rather than guessing a constant runtime distribution for running unknown code).

One specific future work is the automatic scaling down of experiments. Some computational experiments are difficult to independently replicate because they require extensive high-performance computing (HPC) resources. If one could scale down the computational experiment automatically while preserving as much fidelity as possible, anyone could test and verify the scaled-down model [11]. Even if the results are off, this ensures that the experiment has no *logical problem*, and it can be taken to an HPC system with more confidence. This automatic scaling down might even prove useful for the original inquiry to simulate higher fidelity with fewer resources.

Another specific work is to tune the error thresholds on continuous-variable results from computational experiments. Due to the non-associativity of IEEE-754 floating point operations, parallel reductions of the same can return slightly different values. However, setting thresholds on the output in a principled way is an open research question. Dutta et al. does this for machine learning experiments [12], so one might translate their approach to non-deterministic computational experiments. One would need a dataset of runnable computational experiments to evaluate it.

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] 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]

[3] 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]

[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. [Online]. Available: https://dl.acm.org/doi/10.1145/2812803. [Accessed: May 27, 2022]

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

[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 [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

[8] 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

[9] 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]

[10] 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.

[11] A. D. Vu, T. Kehrer, and C. Tsigkanos, “Outcome-Preserving Input Reduction for Scientific Data Analysis Workflows,” Rochester, MI, Oct. 2022 [Online]. Available: https://seg.inf.unibe.ch/papers/ase22.pdf

[12] S. Dutta, A. Shi, and S. Misailovic, “FLEX: Fixing flaky tests in machine learning projects by updating assertion bounds,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Aug. 2021, pp. 603–614, doi: 10.1145/3468264.3468615 [Online]. Available: https://doi.org/10.1145/3468264.3468615. [Accessed: Oct. 13, 2022]