Mitigating failures of software replicability in computational science

Abstract

Irreplicability in computational experiments undermines long-term credibility of science and hinders day-to-day operations of scientists. We plan to use Delta to test computational experiments (often workflows) and study the prevalence and mitigiations of software irreplicability in practice. We will create and share a public dataset of software irreproducibility in computational experiments. This data will be used by us and others to collect data for best practices, make continuous testing feasible, and repair broken programs. These techniques increase the replicability of computational experiments.

Participants

Name	Role	Institution	Email
Daniel S. Katz	PI	NCSA	dskatz@illinois.edu
Darko Marinov	Co-PI	Department of Computer Science, UIUC	marinov@illinois.edu
Reed Milewicz	Extern Collaborator	Sandia National Laboratories	rmilewi@sandia.gov
Samuel Grayson	Student	Department of Computer Science, UIUC	grayson5@illinois.edu

Project Overview

More than 90% of scientists surveyed across all fields use research software and 50% develop software for their research experiments [1]. However, their research software often fails to be replicable[1]. This irreplicability hinders day-to-day

[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

operations, because scientists cannot easily build on each others work. It also undermines trust in science, because scienssts cannot independently scrutinize each others' results.

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" [3], because software with an unstable foundation is analogous to a building with an unstable foundation.

Unfortunately, irreplicability is widespread in the computational science domain. Zhao et al. studied software replicability of computational experiments deposited in the myExperiment registry in 2012 [4]. They found that 80% of the experiments in their selection were irreplicable, 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.

There are many proposed techniques to ensure replicability, of which most fall into two categories: proactive and reactive. A *proactive technique* would control and preserve the environment or application to ensure replicability as software ages, whereas a *reactive technique* would seek to detect and mitigate non-determinism once it occurs. Proactive techniques include using containers (Docker, Singularity, Apptainer, Shifter), virtual machines, system call interposition (Guo's CDE [5], Mozilla's `rr`). Currently, no mainstream proactive techniques can completely control non-determinism due to network resources, pseudorandomness, and parallel program order without sacrificing a large performance penalty or storage costs. The user would be getting possibly irreplicable results without even knowing it.

On the other hand, *continuous testing*[2] seeks to detect rather than eliminate sources of irreplicability. Continuous testing would run the experiment multiple times to assess if the experiment is still producing the same results (replicability). Continuous testing handles the "blind-spots" of proactive techniques. For example, continuous testing might be able to detect when an experiment has non-determinism due to parallel program order. If the user knows that the experiment is non-deterministic, they could experimentally determine the variance of the result or attempt to fix the non-determinism. With proactive-only techniques, the user would not even know that their experiment was non-deterministic.

equivalent result. **Replicable:** one can execute the computational experiment in a different computational environment to get approximately equivalent results. **Reproducible:** one can execute a novel computational experiment to come to the same conclusion Reproducibility is what scientists ultimately want, but repeatability and replicability are necessary stepping stones on the way.

[2] The continuous testing we are proposing here differs from CI/CD because our proposed continuous testing is triggered periodically, while 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 experiments were more likely to break, one could also prioritize testing on that basis, an optimization we term *predictive continuous testing*.

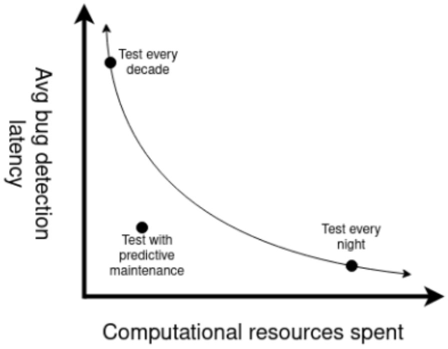

Figure 1: Predicting the rate of software collapse can reduce resource utilization and increase efficacy of continuous testing.

Once a bug has been identified (perhaps by continuous testing), *automated program repair* attempts to apply solutions based on comparing the error-message to a database of common errors and solutions. This technique has been done successfully in other domains [6], and it pairs well with continuous testing, since continuous testing identifies the errors, and automated program repair can try to fix it.

Target Problem

We plan to study the usage and efficacy of these techniques and even improve them. However, to do so, we need data on the replicability of computational experiments, and such data either has not been collected or made public. While many experimental registries exist [7]–[10], **oconner_dockstore_2017?**, they do not store prior results, so we cannot tell if the experiment is replicable. We will collect data on software collapse of computational experiments by automatically running computational experiments from public registries. Then, we will release a public dataset, so other research can proceed from this experiment. The resources of Delta are necessary because there are many registries, each experiment can have many versions, and each version may take a while to run. These registries include:

- nf-core [7]: community-curated, Nextflow pipelines for bioinformatics
- Dockstore [11]: user-contributed, multi-platform (WDL, CWL, Nextflow) pipelines for genomics
- Snakemake Catalog: GitHub-mined, Snakemake pipelines

- WorkflowHub [8]: user-contributed, multi-platform (Galaxy, CWL, Nextflow) pipelines
- myExperiment [9]: user-contributed, multi-platform (Taverna, Galaxy) pipelines for bioinformatics
- WfCommons [10]: researcher-selected, multi-platform (Makeflow, Nextflow, Pegasus) pipelines

Because we cannot take one computational experiment and simulate it one, five, and ten years into the future, we will instead 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 experimental code. Some will still work, and some will fail, due to software collapse. In either case, resulting execution will be stored in our database. The following pseudo-code summarizes this procedure:

```
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,
                ))
```

This dataset will allow us to answer the following research questions (and we will share the dataset so that others can also use it to answer their own research questions in this area of software engineering):

RQ1: What are typical rates of software collapse over time? We plan to reproduce the experiment described by Zhao et al. [4], which assesses if the computational experiments are replicable in our environment. To the existing categorization, we add "repeatable results" as a new column, which says whether two subsequent executions yield the same result. We will also study how the proportion of broken experiments changes with time.

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 collapse for a project based on its history, staleness, and properties of the 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 collapse? Predicting collapse could be useful for institutions, such as national labs, wanting to ensure their computational experiments remain valid while using resources efficiently. 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, for example processor-time and RAM.

RQ5: What are the best practices that improve replicability? We plan to examine choice of workflow manager, lines of code, choice of replicability tools (docker, `requirements.txt` with pinned packages, singularity), and other factors.

RQ6: In what fraction of the cases does automatic repair work? Automatic repair could let one run old experiments off-the-shelf. We can apply techniques similar to Shipwright [6], such as using a language model to categorize many failures into a few clusters.

Description of codes

We developed a Python script that finds computational experiments from the registries and tests them. We use Spack to build our computational environment in `environment.yaml`. We do not require any runtime libraries to be installed at root-level, since Spack can install these at the user-level. We plan to[3] install the Spack environment to network filesystem that all the worker nodes can read. Among other things, our Spack environment contains Python, Singularity, OpenJDK, and common UNIX libraries. The code is not done yet; namely, we need to implement scanning for more registries and execution-handlers for more workflow engines.

We have not explicitly characterized the set of experiments, but we expect they generally consist of a high-level scripting language driving a set of high-performance kernels across a large in-memory dataset. The tasks are usually CPU, with a some experiments having GPU tasks as well. CPU tasks are usually memory-bound, while GPU tasks can be either.

We have yet to parallelize the application, but intend to use parallelism to fully utilize Delta's resources and get our results in a practical time. We can compute multiple experiments simultaneously in a perfectly parallel (also called "embarrassingly parallel") manner. We expect to compute hundreds of experiments simultaneously, each of which takes tens of minutes to terminate; this implies that a global work-queue will not be a performance bottleneck. This system can be implemented easily from our existing code using the parallel-map paradigm in Dask or Parsl.

[3] However, we are open to suggestions.

Experience, readiness, usage plans, and funding sources

We have experience with SLURM batch system, parallel programming, and related HPC technology from using the Campus Cluster.

We do not have estimates on the efficiency of the underlying computational experiments because they are so diverse, and it would take a large HPC resource to gather this efficiency data.

Note that we need to develop more features and robustness in our code before we can run it on an HPC system. First, we need to parallelize (see prior section), then we need to implement more kinds of runners, so that we can run more experiments, then we need to scrape more registries so we have experiments to run. This work can be completed within a month.

Resources required

Registry	Number of experiments
nf-core	32
Snakemake	42
dockstore	~100
WorkflowHub	201
myExperiment	82
WfCommons	7

We have about 500 experiments, 8 versions per experiment, 5 executions per experiment, 1000 core-seconds per execution, which amounts to 5,000 core-hours or 32 cores working for 7 days.

Of these, 5% of experiments have GPU tasks. Therefore, we estimate our tasks require 300 GPU-hours or 2 GPUs working continuously for 7 days.

Each experiment emits about 300 Mb of data. During execution, we need to store the full output so that we can compare their differences. If we use 32 concurrent workers, this gives 10Gb during execution. After the execution, we only need to store the "large" results (see Description of Codes) of the failing experiments, so they can be investigated further. We estimate 25% of the 4,000 executions (i.e., 1,000 executions) will fail, leaving us with 300 Gb. We will likely be able to complete the analysis within a few months.

Resource	Request
Core-hours	5,000
GPU-hours	300
Storage	300Gb for three months

Requested start date and duration

We request to begin execution on December 1. While we expect our allocation to be valid for 1 years, we also expect to use the resources in 2022 Q4, and storage resources in 2022 Q4 and 2023 Q1.

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] 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] 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.

[4] 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

[5] 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

[6] 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.

[7] P. A. Ewels *et al.*, "The nf-core framework for community-curated bioinformatics pipelines," *Nature Biotechnology*, vol. 38, no. 3, pp. 276–278, Mar. 2020, doi: 10.1038/s41587-020-0439-x. [Online]. Available: https://www.nature.com/articles/s41587-020-0439-x. [Accessed: Oct. 31, 2022]

[8] R. F. da Silva, L. Pottier, T. Coleman, E. Deelman, and H. Casanova, "WorkflowHub: Community Framework for Enabling Scientific Workflow Research and Development," in *2020 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS)*, Nov. 2020, pp. 49–56, doi: 10.1109/WORKS51914.2020.00012.

[9] C. A. Goble *et al.*, “myExperiment: A repository and social network for the sharing of bioinformatics workflows,” *Nucleic Acids Research*, vol. 38, no. suppl_2, pp. W677–W682, Jul. 2010, doi: 10.1093/nar/gkq429. [Online]. Available: https://doi.org/10.1093/nar/gkq429. [Accessed: Oct. 31, 2022]

[10] T. Coleman, H. Casanova, L. Pottier, M. Kaushik, E. Deelman, and R. Ferreira da Silva, “WfCommons: A framework for enabling scientific workflow research and development,” *Future Generation Computer Systems*, vol. 128, pp. 16–27, Mar. 2022, doi: 10.1016/j.future.2021.09.043. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X21003897. [Accessed: Oct. 31, 2022]

[11] B. D. O’Connor *et al.*, “The Dockstore: Enabling modular, community-focused sharing of Docker-based genomics tools and workflows,” *F1000Research*, vol. 6, p. 52, Jan. 2017, doi: 10.12688/f1000research.10137.1. [Online]. Available: https://f1000research.com/articles/6-52/v1. [Accessed: Oct. 31, 2022]