A reactive approach to identifying and mitigating software collapse computational science

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 plan to use Delta to test computational experiments (often workflows) in storage to empirically study the nature of software collapse. We will create and share the first public dataset of software decay in computational experiments. This data will be used by us and others to identify 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	National Center for Supercomputing Applications	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	grayson5@illinois.edu

Project Overview

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.

Unfortunately, software collapse is widespread in the computational science domain. Zhao et al. studied software collapse computational of experiments deposited in the myExperiment registry [3]. 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.

There are many proposed techniques to mitigate collapse, of which most fall into two categories: proactive and reactive. A *proactive technique* would control and preserve the environment or application to ensure repeatability as software ages, whereas a *reactive technique* would seek to detect and mitigate nondeterminism once it occurs. Proactive techniques include using Docker, virtual machines, system call interposition [4]. However, at this time no mainstream proactive techniques can completely control non-determinism due to network resources, pseudorandomness, and parallel program order. The user would be getting possibly unrepeatable results without even knowing it.

On the other hand, *continuous testing*[2] seeks to detect rather than eliminate source of non-determinism. Continuous testing would run the experiment multiple times to assess if the experiment is still producing the same results (repeatability). Continuous testing handles the "blind-spots" of proactive testing:

- For non-deterministic pseudorandomness, repeated executions naturally explore the space of possible seeds.
- For parallel program order, repeated executions on different core-counts will explore the space of possible schedules.
- For network resources, repeated executions will identify flaky resources when they become inaccessible. It will not waste the user's time identifying resources that are still accessible, leaving them to focus on only the network resources that are actually flaky.

[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. **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 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]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*.

When an experiment has non-determinism due to parallel program order, for example, continuous testing has a greater chance of detecting this. At least the user would know that the code is non-deterministic, so they could experimentally determine the variance of the result or attempt to fix the non-determinism. 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*.

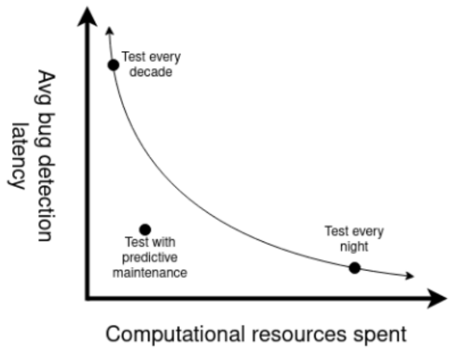

Figure 1: Predicting the rate of software collapse can reduce resource utilization and increase efficacy of continuous testing.

Once a bug has been identified (perhaps by continuous testing), *automated program repair* attempts seeks to apply solutions based on comparing the error-message toa library of common errors and solutions. This has been done successfully in other domains [5]. This pairs well with continuous testing, since continuous testing identifies the errors, and automatic program repair can try to fix it. Continuous testing can help a human encode solutions for errors as well.

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 repeatability of computational experiments, and such data either has not been collected or made public. While many experimental registries exist, they do not store prior results, so we cannot tell if the experiment is repeatable. We will collect data on software collapse of computational experiments by automatically running computational experiments from public registries. 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

- Dockstore
- Snakemake Catalog
- WorkflowHub
- myExperiment
- WfCommons

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 workflow. 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 it 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 replicate the experiment described by Zhao et al. [3], 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.

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, 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 decay? This could be useful for intuitions, 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.

RQ5: What are the best practices that improve replicability? 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: In what fraction of the cases does automatic repair work? Automatic repair could let one run old experiments off-the-shelf. We can apply similar techniques to Shipwright [5], such as using a language model to categorize many failures into a few clusters.

Description of codes

We developed a Python package with a CLI entrypoint that finds computational experiments from the repositories and tests them. We use Spack to build our computational environment in `environment.yaml`. We do not require any run-time 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.

Scanning the registries for versions of experiments constitutes a few HTTP requests and parsing, so the most of the computational time is spent in the underlying computational experiments. 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 take fully utilize Delta's resources and get our results in a practical time. While parallelism usually exists within an experiment, we will mostly exploit parallelism between experiments. That parallelism has low communication and synchronization overhead because the cost of executing one experiment (order of minutes) dominates the cost of fetching and queuing work-items (order of milliseconds). If hundreds of workers were drawing experiments from the queue as soon as they finish, then the central server will receive a request twice every second, which will not cause a bottleneck. We think this can be implemented easily from our existing code using the parallel-map paradigm in Dask or Parsl. The workers need only receive a path or a URL to a version of an experiment to test (less

[3]However, we are open to suggestions.

than a hundred of bytes). The workers need only respond with some statistics about the experiment (hundreds of bytes), and the rest of the output can be dumped onto a network filesystem.

The output can be divided into two parts: the “small” output containing statistics regarding the execution and a hash of the experiment’s output, and the “big” output containing the experiment’s output. Each small-data record of an experiment will fit in hundreds of bytes, so tens of thousands of versions of experiments will yield one megabyte, which will be easily processible on our resources. Research questions 1, 3, 4, and 5 can be answered by just the “small” output. Each big-data record could be on the order of gigabytes. Research questions 2 and 6 require elements from the “big” output of just the collapsed (broken) workfnlows.

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 workflows, 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 and a safety factor of 10, this gives 100Gb 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 10% of the 4,000 executions (i.e., 400 executions) will fail. This leaves us with 120 Gb. We will likely be able to complete the analysis within a few months.

Resource	Request
Core-hours	5,000
GPU-hours	300
Storage	120Gb 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] 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] 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

[4] 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

[5] 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.