Empirical study of decay in computational science

Samuel Grayson, Department of Computer Science grayson5@illinois.edu
Daniel S. Katz, NCSA dskatz@illinois.edu
Darko Marniov, Department of Computer Science marinov@illinois.edu
Reed Milewicz, Sandia National Laboratories rmilewi@sandia.gov

Abstract

More than half of scientists surveyed fields develop software for their research [1]. Unfortunately, the code they develop and use often decays over time. Software decay or "bit rot" is the phenomenon where software, even if unchanged, tends to break as time passes because of its non-obvious dependencies on the execution environment. If computational experiments are allowed to decay, scientists cannot reproduce, independently verify, or build on those results, which frustrates future research and contributes to the ongoing reproducibility crisis in computational science [2] and science more broadly [3].

We want to collect data on decay of computational experiments by automatically running computational experiments from public registries. The dataset we create will store a record for each execution containing its date of publication, the time of execution, compute resources expended in execution, and whether the execution terminated successfully. We will develop predictive models based on the staleness, type of computation, and other determinants to predict the probability that a given workflow will fail. This model will allow us to develop systems that efficiently detect decay, develop repair techniques, and identify best practices. We will publish the dataset and model, because they will be useful to the community beyond this particular work.

Project overview

One way to combat software decay is continuous testing. An automated system can periodically check that the experiment returns the same result. The period should be determiend carefully, because it trades off computational resources with the latency of finding bugs. In order to make informed decisions, one would need to predict the rate of decay for software; one needs to test more frequently if the decay rate is higher. Our model of empirically failures as a function of staleness and other determinants can inform this choice. Then the system will direct its resources to the computational experiments that are most likely to fail.

Once we know what kinds of failure are possible, we can also take steps towards automatic repair. We can apply similar techniques to Shipwright,

such as using a language model to categorize a large number of failures into a small number of clusters **henkel_shipwright_2021?**.

Our dataset should be able to test if certain software practices correspond with a lower empirical failure rate. One of them might be the choice of workflow engine; perhaps some engines have stronger reproducibility guarantees. We will use a "Bayes net" to test for confounding causal variables.

Workflow registries of interest: - nf-core - Dockstore - Snakemake Catalog - WorkflowHub - myExperiment

Workflow engines of interest: - Nextflow - Snakemake - Galaxy - Common Workflow Language - KNIME - OpenWDL

Prior work

Zhao et al. studied the reliability workflows uploaded to the myExperiment registry, finding that 80% were not working **zhao_why_2012?**. Since then, a new generation of workflow engines have emphasized reproducibility, often running each node inside a container, so it would be interesting to reproduce Zhao's study on a modern workflow repository. Zhao et al. only classifies their failures into four categories: volatile third-party resources, missing example data, missing execution environment, and insufficient description about workflows. We would like to use more categories, for example, is the third-party resource a software dependency or data dependency? We would also regress test failures on staleness, which Zhao does not attempt.

Description of codes

We developed a Python package that finds and tests workflows. This package depends on Nix and Docker on the host system. From Nix and Docker, the package can create software environments for each workflow engine of interest. We have yet to parallelize the application, but we think this can be done easily using the parallel-map paradigm in Dask. The code is not done yet; namely, we need to implement scanning for more registries and more workflow engines.

Experience, readiness, usage plans, and funding sources

We have experience with SLURM batch system, Dask programming, and related HPC technology. Note that we need to develop more features and robustness in our code before we can run it on an HPC system.

Resources required

5 registries, 100 workflows per registry, 20 revisions per workflow, 300 CPU seconds per workflow is a total of 3000000 CPU-seconds or 34 CPU-days.

Requested start date and duration

As soon as possible.

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] 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]

[3] 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. [Online]. Available: https://journals.plos.org/plosmedicine/article?id=10.1371/journal.pmed.0020124. [Accessed: Sep. 14, 2022]