Continuously Accelerating Research

Earl Barr
University College London
London, England
e.barr@ucl.ac.uk

Jonathan Bell
Northeastern University
Boston, MA, USA
j.bell@northeastern.edu

Michael Hilton
Carnegie Mellon University
Pittsburgh, PA, USA
mhilton@cmu.edu

Sergey Mechtaev
University College London
London, England
s.mechtaev@ucl.ac.uk

Christopher Timperley
Carnegie Mellon University
Pittsburgh, PA, USA
ctimperley@cmu.edu

***Abstract*—Science is facing a software reproducibility crisis. Software powers experimentation, and fuels insights, yielding new scientific contributions. Yet, the research software is often difficult for other researchers to reproducibly run. Beyond reproduction, research software that is truly reusable will speed science by allowing other researchers to easily build upon and extend prior work. As software engineering researchers, we believe that it is our duty to create tools and processes that instill reproducibility, reusability, and extensibility into research software. This paper outlines a vision for a community infrastructure that will bring the benefits of continuous integration to scientists developing research software. To persuade researchers to adopt this infrastructure, we will appeal to their self-interest by making it easier for them to develop and evaluate research prototypes. Building better research software is a complex socio-technical problem that requires stakeholders to join forces to solve this problem for the software engineering community, and the greater scientific community. This vision paper outlines an agenda for realizing a world where the reproducibility and reusability barriers in research software are lifted, continuously accelerating research.**

***Index Terms*—reproducibility, artifact evaluation, continuous integration, scientific software, containers**

I. Introduction

Reproduction is the cornerstone of science. We use it to ensure that findings generalize. Unfortunately, science suffers from a reproducibility crisis. In 2005, Professor Ioannidis of Stanford's School of Medicine provocatively proclaimed that most published research findings are false [1]. The crisis is ongoing. In a 2016 survey by Nature found that more than 70% of researchers failed to reproduce published results [2] and the website Retraction Watch [3] recorded over 1433 retractions in 2019 [4].

A finding is replicable when the experiment that produced it is meticulously described in enough detail that other researchers can, from its description alone, conduct the experiment and obtain the finding. Experiments — throughout science — increasingly rely on software: for simulation, analyzing data, and conducting experiments. In theses cases, authors should share an *artifact* along with their findings, so that the results can be *reproduced* by running the same software. However: due to the complexity of user interfaces, the abundance of software defects, junior researchers learning on the job, dependencies on evolving libraries, and the wide variety of execution environments, software is often not reproducible. Thus, software reproducibility is a key dimension of the replication crisis that affects science.

Lack of software reproducibility introduces a key problem beyond the traditional problem of unvalidated results: *waste*. To reuse an artifact, researchers are required to spend their valuable time and energy on repetitive, manual tasks rather than focusing on discovery and innovation. Instead of "standing on the shoulders of giants," researchers are required to implement those same efforts over and over again. Over time, these wasted efforts inhibit the pace of research and make it harder for newcomers to enter the field.

Clearly, action is needed. We must accelerate science generally to address societal problems, like climate change and healthcare, for which technical solutions may exist. However, because of software's uptake in science, software reproducibility is an increasingly important component of reproducibility writ large and it is *inherently* a software engineering problem. It orbits core software engineering concerns: software process, documentation, future-proofing, maintainability, efficient execution, and portability. Software engineers are, therefore, best placed to tackle it; indeed, given the stakes, it is our community's duty to rise this challenge and act to make software more reproducible.

Low reusability and reproducibility of software artifacts greatly reduces the pace of scientific research. To continuously accelerate research, we must build infrastructure and implement processes that instill reusability and reproducibility in software artifacts. Engineering reusable software artifacts is not enough. We also must simultaneously incentivize researchers not only to use this infrastructure, but to contribute to its design, development, and deployment.

There have been many efforts to improve the software reproduction crisis within the field of software engineering, such as artifact evaluation. However, these efforts have also exposed new challenges: even artifacts that aim to ensure perpetual reproducibility are subject to decay, and researchers struggle to effectively reuse them. Meanwhile, *continuous integration* (CI) has been widely adopted in industry, allowing

This work was supported in part by the US National Science Foundation grants CCF-2100037 and CNS-2100015

teams to "shift left" on testing by running large test suites far more regularly. Despite its potential to improve reproducibility, CI is not widely adopted in the research community. We propose a research agenda that synergistically improves the software process to tackle the software reproducibility crisis and outline open problems.

II. Reproducibility in Research Software

Neither the reproducibility crisis nor science's increasing reliance on software are new. In response to that, the scientific community formulated four principles, findability, accessibility, interoperability, and reusability, collectively known as FAIR [5], that are applicable to both research data, and also to the algorithms, tools, and workflows that are required to obtain that data. In software engineering, the community introduced artifact evaluation (first at ESEC/FSE in 2011 [6]) to tackle the lack of software reproducibility.

Artifact evaluation is a process to assess the reusability of tools and the reproducibility of experiments that support research articles. This process is now a commonplace process at most software engineering conferences. It has evolved significantly over the past decade, and has also been adopted by many other communities. However, there remain significant challenges: recent surveys have shown that authors and reviewers have differing expectations for artifact construction and evaluation [7], [8]. Moreover, a retrospective analysis of artifacts published in the SE community over the past ten years has *not* shown that artifacts which are evaluated are reused more frequently than those that are not evaluated [9].

Ideally, reusable artifacts should lower the barriers to entry for newcomers to a field. Imagine if a researcher who specializes in genetic algorithms might want to design a new program repair tool. Rather than implement an entire program repair tool and evaluation script themselves, they should be able to *reuse* existing artifacts. Software artifacts might be found by reference in research articles, or in repositories that collect artifacts [10], [11]. However, simply finding a relevant artifact is not sufficient to effectively reuse it, since researchers need to be able to *execute* them at scale. Even in the case of program repair, where a very well-documented evaluation artifact exists [12], it does *not* provide the infrastructure to actually *execute* the artifact using cloud resources. In fact, it is distributed with the following disclaimer: "Warning: the experiment took 313 days of combined execution time." While it is certainly possible to parallelize this evaluation using containerization and cloud computing resources, operationalizing artifacts like these requires specialized distributed systems knowledge that can prevent newcomers from contributing.

At the dawn of artifact evaluation, there was much discussion over what incentives would be necessary to encourage authors to create and share their artifacts. Since then, surveys of authors [8], [13] and post-hoc analyses of bibliometric data [9], [13] have shown that incentives may not be well-aligned. Part of the challenge in building reusable artifacts is that the goals of "reusability" and "repeatability" are often after-thoughts for research prototypes. A recurring suggestion from authors who have embraced these values is to consider these qualities throughout the development of research prototypes [14].

Artifact evaluation processes have focused on how to create an *evaluation* of artifacts for quality attributes like *portability* across computing resources, *reproducibility* of evaluation results and *reusability* of research tools. Portability, reproducibility and reusability are all quality attributes, and, as with most other quality attributes in software engineering, are achieved with the greatest ease when they are considered at each step of the software development lifecycle. Two questions arise: "What does it mean to consider portability, reproducibility, and reusability when engineering research software tools?", and "How can we create tools and processes to make these qualities the de-facto norm?". This manifesto outlines first steps toward answering these questions and issues a call to arms for our community to fully answer them.

III. Ingredients for Reproducible Science

Containers, such as Docker containers [15], can package software with all its dependencies, simplifying sharing and deployment without considerable performance overhead. Containers are widely used in cloud computing [16], continuous integration/delivery [15], and reproducible research [17]. Containers are spawned from images, filesystem snapshots accompanied by configuration files.

Continuous integration (CI) has become a standard industrial practice, allowing unit, integration, end-to-end, and even performance tests to be automatically executed in the cloud. With CI, developers create a fully-automated "workflow" for executing some test suite, leveraging the relatively low cost of cloud computing resources to create a fast feedback loop. For example, MongoDB's CI system automates over 200 different large-scale cloud performance tests that are automatically run, typically once a day, detecting dozens of regressions that are missed by a traditional microbenchmark suite [18].

CI services are especially valuable when it is necessary to design, implement and evaluate several prototypes to better characterize the design space of a solution. While software companies rely on these processes, adoption requires both cloud computing resources and technical know-how to create workflow scripts [19]. Many large development organizations have staff members dedicated to these roles. However, surveys of research software artifact authors and consumers [8] show that researchers building software tools do *not* have the skills or resources to apply CI to their development processes.

Applying this practice in a research setting is a challenge: academia rewards scientific advancement over engineering. Nonetheless, it is vital work that must be done. Rizzi et al examined 26 papers extending the popular KLEE symbolic execution engine [20] and found much duplicated engineering work that raised questions about the soundness of several scientific hypotheses [21].

Our recent ICSE 2022 artifact demonstrated the feasibility of this approach [22], creating a CI evaluation workflow for the Java fuzzer, CONFETTI [23]. We used this CI workflow to debug the upstream project, JQF [24], and reported the fix with an "ICSE publication quality" evaluation in a pull request [25]. Examining this pull request shows the immediate benefits of the approach: we engaged with the project maintainers in a brief discussion of the performance improvement, and each corresponding change is supported by clear empirical evidence. Without the CI workflow, such contributions would be far more complex and time-intensive to evaluate. This CI-enabled contribution is commonplace in the development and maintenance of software in industry, but is shockingly uncommon in academia.

IV. Vision: A Process and Ecosystem for Engineering Reusable Artifacts

We are on the cusp of a revolution in research software and, therefore, software research. Open-source ecosystems like GitHub create a tremendous opportunity for discovering and reusing others' code, and artifact evaluation processes can help ensure that this code is reusable. Indeed, scientists, across many fields, are adopting GitHub. Universities throughout the world are investing heavily in establishing and staffing research development teams to facilitate this transformation[1].

Researchers who have ideas that build on existing research ideas should be able to extend and improve on corresponding software artifacts. Such a process should be frictionless. As this community of reusable artifacts grows, a virtuous cycle will continuously accelerate the research process. However, our current artifact evaluation process incurs substantial overhead for authors and reviewers alike: it will not be feasible to scale this same approach to other scientific venues as-is. This is a complex, socio-technical problem that the entire community stands to benefit from, and which can only be addressed through a coordinated effort.

We propose *CLASSEE*, a community infrastructure and accompanying methodology to continuously accelerate research through Continuous LArge Scale Software Engineering Experimentation. Figure 1 shows how *CLASSEE* supports innovation in software research by providing an infrastructure for automating the execution of software artifacts throughout their lifecycle. *CLASSEE* will leverage the process of continuous integration, creating automated evaluation workflows that run within the GitHub Actions ecosystem. Our design for *CLASSEE* focuses on reusability of *artifacts*, leveraging advanced interfaces for containerization like Modus [26]. This infrastructure will be applicable to any research domain that relies on software.

To enable researchers to efficiently utilize their existing compute resources, *CLASSEE* will provide a publicly-available dashboard for assigning CI workflows to compute resources, allowing researchers to efficiently use these compute services without requiring any specialized training. Once the cloud resources are connected to *CLASSEE* and tied to the researcher's GitHub repository, reproducible evaluations can be automatically triggered. *CLASSEE* will implement a caching layer to ensure continued availability of external dependencies used in an artifact execution, *without* requiring manual effort to identify and archive them. Reviewers auditing the reproducibility of an artifact need only specify the computing resources to be used (e.g. resources belonging to the author, the reviewers or a third party), and await the results.

A. Orchestrating Reuse with Modus

To orchestrate the reproduction of artifacts with *CLASSEE*, we will use a recently proposed language for building container images, Modus [26]. Modus uses logic programming to express interactions among build parameters, specify complex build workflows, automatically parallelize and cache builds, help to reduce image size, and simplify maintenance. In contrast, Dockerfiles, the current dominant solution, force developers to create complex, ad-hoc frameworks, such as the templating approach used in the official Python images [27] just to be able to reuse dependency installation code across several images, which undermines reusability.

Modus expresses build instructions in the form of Datalog rules. Despite the difference in research domains, many common software-focused workflows arise. One example is creating multiple variants of a simulator, of a chemical or physical process, to optimize modeling fidelity. Figure 2 shows another example, drawn from our own research. Here we use Modus to execute program repair experiments. In this example, a program repair tool is executed on the given version of a project to generate a patch in the predicate `patch`. This patch is then applied to a fresh version of the project and is tested in the predicate `test`. The used predicates such as `checkout` and `install_tool` can be either defined by the user or provided by the developers of tools, which helps on-boarding.

The key advantage of the definitions presented in this example is their modularity — the predicates abstract away irrelevant information such as tool installation instructions and temporary directories, and can be transparently reused in other experimental scripts. Apart from that, the side effect of each predicate is stored into a separate image layer, which enables automatic caching and parallelization. This is important for research experiments, since executing experiments takes a significant amount of time and computing resources.

Reproducibility is not free, it requires continuous maintenance; it is, in fact, prohibitive for a single researcher. Distributing this work amortizes the cost, but undermines reproducibility, as discrepancies and divergences in the artifacts arise. To address this problem, we will design a flexible module import system, which, combined with Modus' modularity, will enable users to share and reuse their artifacts, build scripts and experimental workflows in a transparent and convenient way. The import system will make our infrastructure decentralized: individual components will be maintained by independent groups of researchers, while still preserving integrity and reproducibility of the infrastructure as a whole.

[1]Examining the first 50 hits from a Google search for "research software development teams in universities" at the time of this writing reveals that ca 90% of these hits concern just such initiatives, of which https://www.ucl.ac.uk/advanced-research-computing/expertise/research-software-development at University College London is a representative example.

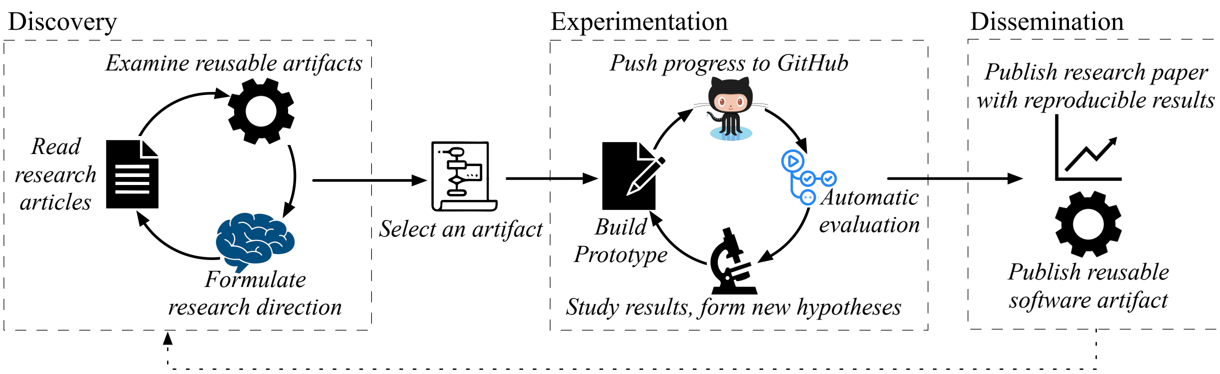

Fig. 1. Workflow for building and evaluating software tools with *CLASSEE*: researchers read research articles, examine reusable artifacts, and formulate new research directions. Building on those existing artifacts, scientists can automatically execute complex evaluations of those artifacts. Reproducibility is then "baked in" to the new artifact, which can be easily shared and adopted by others to discover.

```
base(project, version, "/experiment") :-
  from("ubuntu:20.04"),
  install_project_dependencies(project),
  checkout(project, version, "/experiment").

patch(project, version, tool, patch_file) :-
  base(project, version, directory),
  install_tool(tool),
  tool_config(tool, project, config),
  run_tool(tool, config, directory, patch_file).

validate_patch(project, version, tool) :-
  base(project, version, dir),
  patch(project, version, tool, patch_file)
    ::copy(patch_file, patch_file),
  run(f"cd ${dir} && patch -p0 -s < ${patch_file}"),
  test_project(project, version, dir).
```

Fig. 2. Modusfile defining program repair experiments with GenProg using reusable images and layer-building functions.

The decentralized architecture will promote a wider participation in research community, and will reduce the burden of researchers who currently have to maintain custom variants of a large number of third-party tools and benchmarks to make their research reproducible.

GitHub Actions' eponymous *action* command encapsulates a re-usable step that might be performed by many different workflows. To exploit this functionality, we will design, implement and document reusable actions based on Modus predicates that perform tasks common to many software tool evaluations. We will work with the community using established human-centered design methodologies [28] to create standardized interfaces for invoking tools on common datasets, as well as standardized output formats for those tools to generate. These actions will make it easier for researchers building entirely new evaluation workflows to benefit from common implementations of core actions including: (1) Caching and replaying all external network requests; (2) Monitoring experiment execution and gathering real-time telemetry; (3) Invoking popular software artifact datasets like BugSwarm [29], Defects4J [30] and Bugs.Jar [31]; (4) Generating evaluation reports using tools like Jupyter Notebook and R-Markdown; (5) Invoking cluster management tools to launch and teardown cloud resources.

B. Core, Community Infrastructure

We also imagine that several core, key infrastructure components will be useful for all artifacts, regardless of how they are constructed. We envision deploying this core infrastructure as a public, community service, also allowing research to self-hosting it if preferred.

1) Caching and Reproducibility: Software artifacts are typically distributed without the third-party dependencies that are needed to compile and execute them. For example: while the software artifact dataset *BugSwarm* containerizes each artifact in a docker image for ease of reproducibility, many of the artifacts do not include *all* external dependencies, which require manual efforts to resolve [29]. Hence, we have designed a caching service for *CLASSEE* to improve the performance of *CLASSEE* by lowering network utilization, while simultaneously ensuring the continued availability of those dependencies. This service will be build atop the popular, open-source Squid proxy cache, which can be configured to locally cache all HTTP and HTTPs traffic and to later serve all requests from that cache [32], [33]. Squid supports an "offline" mode, which, when set, will respond to queries *only* from its cache. By using a self-signed root CA, Squid can even be used to cache and intercept encrypted HTTPs traffic [33]. We will create a containerized Squid deployment that is pre-configured to work with *CLASSEE* to archive all external dependencies for each CI workflow execution. This tool will be directly integrated with CI workflows through a re-usable GitHub Action. We will archive the cache along with the artifact that generated it; to reproduce the artifact, the proxy server will have its cache pre-populated in "offline" mode.

2) CLASSEE *CI Runner Service:* GitHub Actions' architecture is designed around a cloud service that coordinates the execution of CI workflows on "runners" — machines that can be scaled up or down, each of which runs an entirely self-contained build task. Although the service places a limit on the number of minutes of *cloud* runners (provided by GitHub) that each project can use for free, developers can deploy "self-hosted runners' on their existing compute resources and use the platform for free. *CLASSEE* will provide a seamless bridge between GitHub Actions and cloud computing resources that are available to researchers (including specialized hardware like GPUs), entirely automating the provisioning of CI runners to low or no-cost resources.

3) Documentation and Training: We are sure that many researchers will want to create different evaluation workflows, or to integrate tools that we could not have imagined — a significant aspect of *CLASSEE* will include the development, evaluation and dissemination of training materials to help researchers adapt and re-use the CI components that we will develop as part of this project. Working in collaboration with community stakeholders, we will create, document and share reusable CI workflows to automate the execution of common large-scale software tool evaluations.

V. Future Plans

Our future plans build on our prior work, including the CONFETTI GitHub Actions artifact [22] and Modus [26]. However, these pieces must still be brought together. Our immediate plans are to develop *CLASSEE*'s core community infrastructure described in Section IV. We will create documentation and training materials to facilitate on-boarding to *CLASSEE*, and provide a publicly-hosted installation of *CLASSEE* free of charge. We plan to work with the program repair, regression testing, and fuzzing communities to build template workflows for conducting reusable, large-scale evaluations of these tools.

Building off of the ACM SIGSOFT Empirical Research Standards [34], we plan to engage closely with the software engineering community to create and share best practices for conducting large-scale software evaluations. We expect that some aspects of these best practice discussions will be broadly applicable, for example: determining how to sample a subset of an evaluation for a smoke test, how to avoid over-fitting a tool to an evaluation dataset, how to ensure reproducibility and how to mitigate the effects of non-determinism.

Our framework will provide a foundation for other tooling aimed at improving research, like Planalyzer [35], Soylent [36], and large language models like chatGPT[2].

Ensuring success of such a large-scale project will require continuous evaluation. Thankfully, some of the evaluation processes can be entirely automated, perhaps even using *CLASSEE* itself. For example: we plan to create CI workflows that deploy a testing instance of *CLASSEE*, and then execute common workflow templates, effectively using the platform to test itself. We will collect various quantitative metrics from those workflow executions including the time spent, the reproducibility of the result, and the overall performance of the platform including error rates and throughput. We imagine that we would regularly run only a subset of the workflows, but will also plan to evaluate these metrics for *all* of the workflow templates and artifacts at least quarterly, so as to detect otherwise un-noticeable regressions. To evaluate the utility and usability of the tool, we plan to use surveys, interviews and observational studies to identify opportunities to improve uptake and facilitate sustained adoption.

We plan to evaluate our training and curricular materials by applying them in our own classes, sharing them with the community, and using robust education research methods [37] to inform the iterative improvement of these materials. We will recruit students to work on semester-long projects, developing and using *CLASSEE*. While providing a valuable training opportunity for the students to learn about cutting-edge software engineering technology, these students will provide direct and useful feedback to improve our materials. We plan to make all of these materials available at the project website, https://www.classee.cloud/.

[2]chatGPT website: https://openai.com/blog/chatgpt/

VI. Conclusion

Scientific research faces a software crisis of its own: we build software for experimentation and to validate hypotheses, but creating *reusable* and *reproducible* software is a tremendous burden. However, there is so much benefit to a world in which research articles are accompanied by software artifacts that are truly reusable in the sense that another researcher could modify and re-execute them. A decade's worth of artifact evaluation processes have shown that while it is possible to create reusable artifacts, authors benefit most from a process that instills reusability and reproducibility from the inception of a project to its publication. A community infrastructure that brings continuous integration to research software will serve as a first step towards creating an ecosystem of truly reusable artifacts.

References

[1] J. P. Ioannidis, "Why most published research findings are false," *PLoS medicine*, vol. 2, no. 8, p. e124, 2005.

[2] M. Baker, "1,500 scientists lift the lid on reproducibility," *Nature*, vol. 533, no. 7604, 2016.

[3] "Retraction Watch," https://retractionwatch.com/, accessed: 2022-10-12.

[4] "The top retractions of 2019," dec 2019.

[5] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne *et al.*, "The fair guiding principles for scientific data management and stewardship," *Scientific data*, vol. 3, no. 1, pp. 1–9, 2016.

[6] S. Krishnamurthi and J. Vitek, "The real software crisis: Repeatability as a core value," *Commun. ACM*, vol. 58, no. 3, pp. 34–36, feb 2015. [Online]. Available: https://doi.org/10.1145/2658987

[7] B. Hermann, S. Winter, and J. Siegmund, "Community expectations for research artifacts and evaluation processes," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 469–480. [Online]. Available: https://doi.org/10.1145/3368089.3409767

[8] C. S. Timperley, L. Herckis, C. L. Goues, and M. Hilton, "Understanding and improving artifact sharing in software engineering research," *Empir. Softw. Eng.*, vol. 26, no. 4, p. 67, 2021. [Online]. Available: https://doi.org/10.1007/s10664-021-09973-5

[9] S. Winter, C. S. Timperley, B. Hermann, J. Cito, J. Bell, M. Hilton, and D. Beyer, "A retrospective study of one decade of artifact evaluations," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE, 2022.
[10] Software Heritage Foundation, "Software heritage," https://www.softwareheritage.org.
[11] L. Arumugam, V. N. Subramanian, and M. Nagappan, "Segarage: A curated archive for software engineering research tools," *SIGSOFT Softw. Eng. Notes*, vol. 44, no. 3, p. 13, nov 2019. [Online]. Available: https://doi.org/10.1145/3356773.3356777
[12] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts," in *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*, 2019. [Online]. Available: https://arxiv.org/abs/1905.11973
[13] R. Heumüller, S. Nielebock, J. Krüger, and F. Ortmeier, "Publish or perish, but do not forget your software artifacts," *Empirical Softw. Engg.*, vol. 25, no. 6, pp. 4585–4616, nov 2020. [Online]. Available: https://doi.org/10.1007/s10664-020-09851-6
[14] N. Zilberman and A. W. Moore, "Thoughts about artifact badging," *SIGCOMM Comput. Commun. Rev.*, vol. 50, no. 2, pp. 60–63, may 2020. [Online]. Available: https://doi.org/10.1145/3402413.3402422
[15] D. Merkel *et al.*, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.
[16] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: a state-of-the-art review," *IEEE Transactions on Cloud Computing*, vol. 7, no. 3, pp. 677–692, 2017.
[17] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.
[18] H. Ingo and D. Daly, "Automated system performance testing at mongodb," in *Proceedings of the Workshop on Testing Database Systems*, ser. DBTest '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3395032.3395323
[19] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 426–437. [Online]. Available: https://doi.org/10.1145/2970276.2970358
[20] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756
[21] E. F. Rizzi, S. Elbaum, and M. B. Dwyer, "On the techniques we create, the tools we build, and their misalignments: A study of klee," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 132–143. [Online]. Available: https://doi.org/10.1145/2884781.2884835
[22] J. Kukucka, L. Pina, P. Ammann, and J. Bell, "CONFETTI: Amplifying Concolic Guidance for Fuzzers," https://figshare.com/articles/software/CONFETTI_Amplifying_Concolic_Guidance_for_Fuzzers/16563776, 1 2022.
[23] ——, "Confetti: Amplifying concolic guidance for fuzzers," in *Proceedings of the 2022 International Conference on Software Engineering*, ser. ICSE, 2022.
[24] R. Padhye, C. Lemieux, and K. Sen, *JQF: Coverage-Guided Property-Based Testing in Java*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 398–401. [Online]. Available: https://doi.org/10.1145/3293882.3339002
[25] J. Bell, "Faster, collision-free coverage instrumentation #171," https://github.com/rohanpadhye/JQF/pull/171, 2022.
[26] C. Tomy, T. Wang, E. T. Barr, and S. Mechtaev, "Modus: A datalog dialect for building container images," 2022.
[27] "Docker official image packaging for python," https://github.com/docker-library/python, 2022, accessed: 2022-10-01.
[28] T. Farooqui, T. Rana, and F. Jafari, "Impact of human-centered design process (hcdp) on software development process," in *2019 2nd International Conference on Communication, Computing and Digital systems (C-CODE)*. IEEE, 2019, pp. 110–114.
[29] C. Rubio Gonzalez, "Bugswarm website," https://bugswarm.github.io.
[30] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440. [Online]. Available: http://doi.acm.org/10.1145/2610384.2628055
[31] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs.jar: A large-scale, diverse dataset of real-world java bugs," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 10–13. [Online]. Available: https://doi.org/10.1145/3196398.3196473
[32] Squid Project, "Squid: Optimising web delivery," http://www.squid-cache.org, 2022.
[33] ——, "Squid configuration directive ssl_bump," http://www.squid-cache.org/Doc/config/ssl_bump/, 2022.
[34] P. Ralph, "Acm sigsoft empirical standards released," *SIGSOFT Softw. Eng. Notes*, vol. 46, no. 1, p. 19, jan 2021. [Online]. Available: https://doi.org/10.1145/3437479.3437483
[35] E. Tosch, E. Bakshy, E. D. Berger, D. D. Jensen, and J. E. B. Moss, "Planalyzer: Assessing threats to the validity of online experiments," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–30, 2019.
[36] M. S. Bernstein, G. Little, R. C. Miller, B. Hartmann, M. S. Ackerman, D. R. Karger, D. Crowell, and K. Panovich, "Soylent: a word processor with a crowd inside," in *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, 2010, pp. 313–322.
[37] W. Luttrell, *Qualitative educational research: Readings in reflexive methodology and transformative practice*. Routledge, 2010.