Continuously Accelerating Research

***Abstract*—Scientific research is faces a software crisis. Software powers experimentation, and fuels insights, yielding new scientific contributions. Yet, the research software that we develop is often difficult for other researchers to reproducibly run. Even if research results can be reproduced, creating research software that is truly reusable, and can be easily extended by other researchers. As software engineering researchers, we believe that it is our duty to create tools and processes to instill these qualities of reusability and reproducibility in research software throughout the development process. This paper outlines a vision for a community infrastructure that will bring the benefits of continuous integration to scientists developing research software. This approach will appeal to researcher's intrinsic self-motivations by making it easier to develop and evaluate research prototypes. This 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 as a whole. This vision paper outlines an agenda to realize a world where the reproducibility and reusability barriers in research software are lifted, continuously accelerating research.**

I. Introduction

Reproduction is the cornerstone of science. We use it to ensure that findings generalize. Unfortunately, science is currently suffering 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. In this case, authors might 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, 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.

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 giants 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 problems, like climate change, 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, the 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. At the same time, new software process paradigms like *continuous integration* (CI) have been widely adopted in industry, allowing teams to "shift left" on testing by running large test suites far more regularly. Despite its potential to improve reproducibility and reusability, CI is not widely adopted in the development of research software. We propose a research agenda that synergistically improves the software process to tackle the software reproducibiilty crisis and outline open problems.

II. Reproducibility in Software Engineering

Neither the reproducibiliity crisis or science's increasing reliance on software are new. So, it is not surprising that our community introduced artifact evaluation (first at ESEC/FSE in 2011 [5]) 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. The process 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 [6], [7]. 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 [8].

Ideally, reusable artifacts should lower the barriers to entry for newcomers to a field. For example: imagine if a researcher who specializes in genetic algorithms might want to design a new tool to automatically synthesize patches to repair defects in software. 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 [9], [10]. 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 [11], 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 [7], [12] and post-hoc analyses of bibliometric data [8], [12] 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 [13].

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

a) Containers:: Containers, such as Docker containers [14], can package software with all its dependencies, simplifying sharing and deployment without considerable performance overhead. Containers are widely used in cloud computing [15], continuous integration/delivery [14], and reproducible research [16]. Containers are spawned from images, filesystem snapshots accompanied by configuration files. Images, according to a widely-used OCI specification [17], consist of layers that store changes to the underlying filesystem, such as file additions, modifications and deletions, that are combined in runtime using a union mount filesystem.

b) Continuous Integration:: In the past decade, 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. CI serves as a force-multiplier for developers' time, allowing developers to focus on writing code, rather than on deploying and running large test suites. 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]. Workflow executions can be triggered when code is pushed to a specific branch of a repository, when a pull-request is opened on a repository, or through external or manual triggers. When a workflow is executed, its output linked to the revision of the code executed, ensuring traceability.

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 large and small 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, as identified by surveys of research software artifact authors and consumers [7], 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].

A recent ICSE 2022 artifact demonstrated the feasibility of this approach [22], creating a CI evaluation workflow for the Java fuzzer, CONFETTI [23]. The authors 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: the authors and contributors engaged 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.

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

We are on the cusp of a revolution in 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

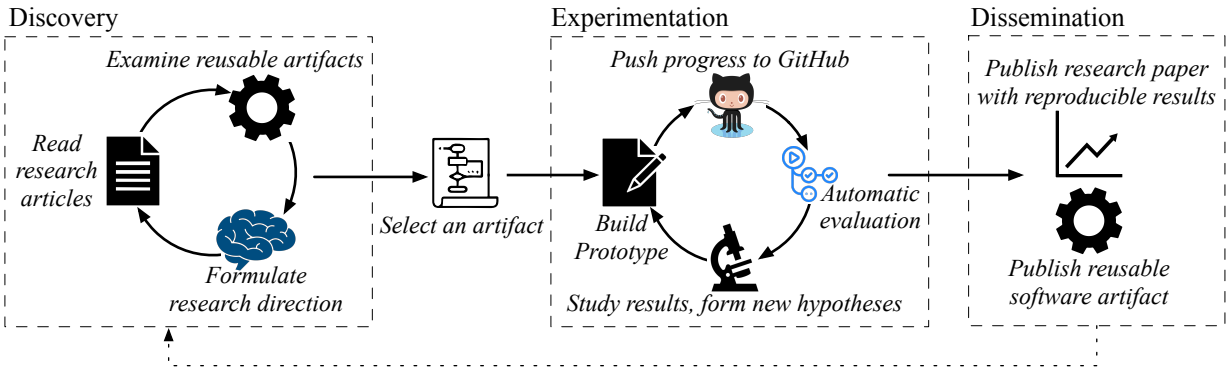

Fig. 1. Workflow for building and evaluating software tools with *CLASSEE*: researchers read research articles, examine reusable artifacts, and formulate new research directions. Building on those existing artifacts, scientists can automatically execute complex evaluations of those artifacts. Reproducibility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

reusable. Researchers who have ideas to extend and improve on existing research ideas should be able to extend and improve on corresponding software artifacts. Such a process should be frictionless, requiring no significant additional efforts on the part of authors constructing these artifacts or the reviewers evaluating them. As this community of reusable artifacts grows, a virtuous cycle will continuously accelerate the research process, feeding new discoveries. 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. 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 recent insights from the software engineering community [22], creating automated evaluation workflows that run within the GitHub Actions ecosystem. Our design for *CLASSEE* focuses on reusability *of artifact components*, leveraging advanced interfaces for containerization like Modus [26]. This infrastructure will be applicable to any research domain that relies on software.

CLASSEE will be a community infrastructure to support researchers who build, evaluate, and share software tools. To make use of *CLASSEE*, researchers can begin by forking an existing project, or initializing a new project using starter workflow files. 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* . 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. Reusable Containerized Components

As described in Section III, many core components for creating reusable and reproducible artifacts already exist. The key challenge is in designing and integrating the necessary abstractions so that the effort that goes into building one artifact can be leveraged in the construction of another (perhaps quite unrelated) artifact. In addition to providing an ecosystem of reusable artifacts, *CLASSEE* will organize a collection of reusable components, making it easier to bootstrap new artifacts without starting from scratch.

1) Composable Containers: Modus [26] is a language for building container images. Side effects of Modus' scripts are segregated into layers of OCI container images [17], which enables automatic caching, parallelisation and reuse. Modus' underlying formalism, Datalog, enables concise definition of complex parameterized builds. These properties make Modus an excellent fit for defining experimental infrastructure, since it facilitates efficient use of heterogeneous execution environment in a reproducible fashion.

2) Composable Actions and Workflows: GitHub Actions provides the novel abstraction of an *action*, which encapsulates a re-usable step that might be performed by many different workflows. We will design, implement and document reusable actions that perform tasks common to many software tool evaluations. We will work with the community using established human-centered design methodologies [27] 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 [28],

Defects4J [29] and Bugs.Jar [30]; (4) Generating evaluation reports using tools like Jupyter Notebook and R-Markdown; (5) Invoking cluster management tools to launch and teardown cloud resources; and. We will use these actions as scaffolding to create reusable template workflows to support evaluations for tools that address common software engineering problems like: (1) Regression testing; (2) Fuzzing and test generation; and (3) Automated program repair;.

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 Reproducibiltiy: 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 [28]. 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 [31], [32]. 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 [32]. 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

We have seen developments begin in the community that support our vision, such as the CONFETTI GitHub Actions artifact [22] and Modus Prolog dialect for specifying docker images [26]. However, these pieces must still be brought together into a core piece of reusable infrastructure. 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 [33], 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. Other aspects, such as the interfaces used to invoke a tool by a workflow or to process its output and visualize key performance indicators, might be specialized to communities, but we nonetheless expect that the processes that we follow to identify and refine these norms will be useful for others.

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. It is the duty of the software engineering community to rise to meet these challenges — to improve our own software artifacts, and to transfer these insights to the greater scientific community. 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] 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

[6] 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

[7] 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

[8] 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.

[9] Software Heritage Foundation, "Software heritage," https://www.softwareheritage.org.

[10] 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

[11] 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

[12] 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

[13] 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

[14] D. Merkel *et al.*, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[15] 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.

[16] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[17] "Open container initiative," https://opencontainers.org/, 2022, accessed: 2022-10-01.

[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] 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.

[28] C. Rubio Gonzalez, "Bugswarm website," https://bugswarm.github.io.

[29] 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

[30] 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

[31] Squid Project, "Squid: Optimising web delivery," http://www.squid-cache.org, 2022.

[32] ——, "Squid configuration directive ssl_bump," http://www.squid-cache.org/Doc/config/ssl_bump/, 2022.

[33] 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