Getting Research Software to Work: A Case Study on Artifact Evaluation for OOPSLA 2019

Erin Dahlgren
Accelerate Publishing[1]

Abstract— **Due to new peer-review programs, researchers in certain fields can now receive badges on their papers that reward them for writing functional and reusable research code. These badges in turn make their research more attractive for others to cite and build upon. Unfortunately, some submissions to these new programs do not pass the lowest bar, and many submissions are difficult for reviewers to simply setup and test. To understand how to improve submissions and how to help researchers gain badges, we studied the artifact evaluation process of OOPSLA 2019, an ACM conference on the analysis and design of computer programs. Based on reviewer experiences, we highlight best practices and we discuss whether guidelines, tools, or larger cooperative efforts are required to achieve them. To conclude, we present ongoing and future work that helps researchers share and use research code.**

1. Introduction

Many researchers today are frustrated with how difficult it is to reproduce published results [1], [2]. Despite the widespread use of software to conduct research [3], rarely can research software be found, run, and reused, making important research results hard to trust and build upon [4]. In an effort to address this, the Association of Computing Machinery (ACM) created an initiative to award badges to research papers backed by functional, reusable research code that reproduces results [5]. These badges make research papers stand out as more transparent and credible, potentially gaining authors more citations from their peers.

In this report, we present a case study on a subset of the research software artifacts submitted to OOPSLA 2019, an ACM conference on the analysis and design of computer programs [6]. To better understand how to help researchers submit quality artifacts and to better understand how to improve the review process, we collected data on submitted artifacts, on documentation, and on peer-reviewer experiences.

This report is organized as follows: Part 2 describes the artifacts we studied, the methods we used, and the categories of data we present in Parts 3-4. Parts 3-4 presents the data we collected to answer the questions "What makes an artifact hard to test?" and "What makes an artifact easy to test?". Part 5 summarizes and discusses the data in Parts 3-4, and finally, Part 6 presents ongoing and future work. Henceforth, the terminology and acronyms below will be used interchangeably through this report:

Term	*Description*
artifact	research software artifact
reviewers	members of an artifact evaluation committee
image	contains the files of an artifact
archive	a compressed directory
VM	short for "Virtual Machine"
container	short for "Linux container"
open source	freely readable code
code	software

2. Methodology

To collect data, we participated as a member of the artifact evaluation committee for the OOPSLA 2019 conference [6]. Since OOPSLA accepts research on the analysis and design of computer programs, naturally in many cases research artifacts were in and of themselves the research results, serving as examples of the designs described in papers.

Of the 44 artifacts reviewed by the committee, 28 (64%) opted into being observed by us for the purpose of this study. For each opt-in, we were able to view a draft of a research paper that explained the research results, an artifact, a "Getting Started" guide written by the artifact's authors, and comments made by other reviewers on topics ranging from setting up artifacts, testing functionality, interpreting documentation, to the reusability of code.

As a member of the committee, we also tested 3 artifacts in-depth, meaning that we attempted to setup and execute these artifacts, and we checked whether they produced results that could be compared with those reported in their papers. We did not evaluate the correctness of the results because we lacked sufficient domain expertise.

While the committee evaluated artifacts along a number of dimensions [5], in this report we only present data on the review of *functionality*. Our focus is due to the fact that functionality is a necessary precondition of other dimensions like code reusability and reproducible results. Furthermore, there were enough functional issues with artifacts to warrant a closer look: for 24/28 artifacts we observed, reviewers faced issues setting up and executing them.

While the number of artifacts we reviewed is low, this

[1]Accelerate Publishing is a small non-profit that works with artifact evaluation committees to improve review processes and to help more researchers submit their software for review. It specifically develops reports, software tools, and collaborative projects that help both artifact authors and reviewers. Erin Dahlgren is a researcher and software developer at Accelerate Publishing. Contact: `edahlgren@uchicago.edu`

[2]This work is licensed as Free Culture Work under the Creative Commons CC-BY-4.0 license. See the "License" section for more detail.

report is still valuable, since many of the issues we observed could lead one to believe that an artifact doesn't work at all. And the lost value of even a single artifact matters: the weeks to months required to reproduce its behavior is evidence enough. We therefore encourage the reader not to view the data we present as evidence of the relative importance of an issue. An issue that happens only once may be the very thing that makes an artifact impossible to use. Instead, view the data in the following sections as a detailed picture of how researchers and reviewers need more help.

We analyzed the data as follows: First, we collected and separated all of the negative and positive comments made by the reviewers of the 28 artifacts. We then grouped similar comments into "types", and we grouped those types of comments into high-level categories. Below are the categories that resulted from this process. Color-coding is used consistently to refer to these categories throughout this report.

● Environment	Related to the operating system, software dependencies, and physical resources needed by an artifact
● Format	Related to how an artifact is packaged and distributed, for example, as a VM, container, archive, or any other format
● Content	Related to an artifact's code, documentation, helper scripts, and tests
● Execution	Related to compiling or executing an artifact

In Parts 3-4, we present a combination of quantitative data (the number of artifacts that reviewers commented on negatively or positively) and qualitative data (details from reviewer experiences). This combination of data makes it possible to spot trends and nuance.

3. What makes an artifact hard to test?

Table 1 shows the types and frequencies of negative comments made by reviewers while setting up and executing artifacts. The types of comments are ranked by their frequency in reference to artifacts below:

Artifacts with at least one type of comment below	24/28
● Long running tests	15/28
● Not enough resources	10/28
● Problems with docs	10/28
● Issues compiling or running	8/28
● Issues with VM or container	5/28
● Ignored errors	5/28
● Issues with software dependencies	4/28
● Works in limited environments	3/28
● Errors in scripts	2/28
● Too complicated	2/28
● Downloads during execution	2/28

Grid 1 provides more context to Table 1 with details from reviewer experiences. Of note are major frustrations like trying and failing repeatedly to find the right software dependencies, and needing to run experiments for 8 times longer than expected.

4. What makes an artifact easy to test?

Table 2 shows the types and frequencies of positive comments, similar to Table 1. The types of comments are ranked by their frequency in reference to artifacts below:

Artifacts with at least one type of comment below	7/28
● Self-contained	4/28
● Comprehensive documentation	4/28
● Lightweight	1/28

The fewer number of topics (compared to Part 3) is due to the fact that there were far fewer positive comments than negative ones. Understandbly, reviewers may have been more motivated to comment when frustrated, or out of need.

Further, while it was not easy to objectively assess "comprehensive documentation" or how "lightweight" an artifact appeared to be, it was possible to objectively count how many artifacts were self-contained in a VM or container. Encouragingly, the vast majority (86%), were self-contained in one of these ways:

● Artifacts contained in a VM	14/28
● Artifacts contained in a container	10/28

Grid 2 provides more context to Table 2 with details from reviewer experiences. In general, reviewers seemed to appreciate artifacts that were flexible and understandable, with minimal setup required.

5. Discussion

This section highlights best practices that prevent the issues reviewers encountered in Part 3 and result in the beneficial characteristics of Part 4.

First, an artifact that is a complete snapshot of its dependencies (data, software libraries, and code) is relatively easy for reviewers to test because setup steps have been minimized. For example, to test a complete snapshot of an experiment stored in a container, only two setup steps are required: (1) download the container image, (2) mount it and attach a shell.

Further, a complete snapshot is relatively easy to run remotely where there may be more resources available, for example in a high-performance cluster or in a cloud environment. This is because by definition a complete snapshot is self-contained. In other words, one can move the complete snapshot easily from place to place instead of manually setting up dependencies, data, and code in each new location.

It must be noted that authors may have a good reason to download large dependencies on the fly. If, for example, a data set is extremely large, storage requirements can be minimized by fetching and caching parts of the data set as needed. This may be fine if the fetched content is verifiable, checked, and always available. But in practice, these best practices are difficult to check and uphold. A compromise for authors and reviewers may be to have data packaged separately. For example, a container image containing a large

Table 1: Types and frequencies of negative comments

Category	*Type of comment*	*Artifacts with type of comment*
● **Environment**	**Not enough resources.** Reviewers didn't have enough physical resources to test the artifact locally in a reasonable amount of time.	10/28
	Issues with software dependencies. Reviewers struggled to find and install the right software dependencies, sometimes preventing them from setting up and testing the artifact.	4/28
	Works in limited environments. Reviewers had difficulty getting access to proprietary operating systems and running benchmarks locally.	3/28
● **Format**	**Issues with VM or container.** Reviewers encountered errors when they tried to setup VM's and containers, or the VM's and containers slowed down the review process.	5/28
● **Content**	**Problems with docs.** Reviewers encountered typos in instructions and missing or unclear instructions.	10/28
	Errors in scripts. Reviewers wasted time debugging typos and unexpected errors in helper scripts.	2/28
	Too complicated. Reviewers struggled to complete complicated instructions and to test complicated artifacts.	2/28
● **Execution**	**Long running tests.** Reviewers struggled to complete tests that took hours or days.	15/28
	Issues compiling or running. Reviewers encountered errors when they tried to compile or run the artifact.	8/28
	Ignored errors. Reviewers weren't confident about artifacts that emitted errors, even if the results produced were correct.	5/28
	Downloads during execution. Reviewers spent a long time running test cases that downloaded data on the fly. Reviewers were also worried that such data would not always be available.	2/28

Table 2: Types and frequencies of positive comments

Category	*Type of comment*	*Artifacts with type of comment*
● **Format**	**Self-contained.** Reviewers were enthusiastic about artifacts that required minimal setup and worked seamlessly.	4/28
	Lightweight. Reviewers benefited from artifacts that required minimal storage space. They also appreciated being able to download small parts of a large the artifact.	1/28
● **Content**	**Comprehensive documentation.** Reviewers praised clear, easy to follow documentation that covered most if not all aspects of the artifact.	4/28

Grid 1: Context for types of negative comments

● **Not enough resources.** In one case, a container image was so large that it caused the reviewer's machine to run out of inodes. In another case, the short version of an experiment (3 hours) ended up taking a full day. For this artifact, reviewers were unsure how long to wait before assuming the artifact wasn't functional. For the same artifact, a reviewer also struggled to produce results when the artifact failed with an "out of space" error. A different artifact relied on the network so much that one reviewer had trouble finishing tests on a spotty internet connection. In 7/10 of the cases, reviewers only ran a subset of functionality. In 4/10 cases, reviewers reported needing to run experiments for multiple days and for only 2/10 cases did reviewers report they were able to do so.

● **Issues with dependencies.** In two cases, software dependencies were not precisely documented. Reviewers then went through a time-consuming process of trial and error in their attempt to find the right software to install. In another case, software dependencies and precise version numbers *were* documented but they weren't tested ahead of time and they happened to be wrong. Reviewers therefore could not setup the artifact properly until the artifact's author corrected the mistake. In a different case, it was unclear to reviewers *how* to install dependencies, even though they were documented.

● **Works in limited environments.** In two cases, reviewers tried to extract an artifact from a VM to set it up locally. Both of these attempts failed due to problems with dependencies or the artifact itself. In another case, reviewers were told to test an artifact only on a Windows operating system. Some reviewers did not have access to this environment and could not test it this way.

● **Issues with a VM or container.** In one case, reviewers struggled with a container that ran expectionally slow on macOS, due to how file mounts was implemented on that platform. In a different case, reviewers struggled to use a VM that had a small (4 inch) display. In 3/5 cases, the issues involved *launching* VM's and containers.

● **Problems with docs.** In 6/10 cases, reviewers encountered typos; in 5/10 cases, reviewers noticed that instructions were missing; and in 2/10 cases, instructions were present but they were confusing or unclear. Reviewers were often able to fix the typos but many struggled with the other issues.

● **Errors in scripts.** In one case, a script containing errors downloaded wrong software dependencies. Despite frustration, in all cases reviewers were able to debug the errors in scripts.

● **Too complicated.** In one case, an artifact had so many manual steps that reviewers inevitably made accidental mistakes. One reviewer lost so much time during this process that they didn't have time to re-execute the instructions correctly. In a different case, the complicated structure of an artifact made running its benchmarks cumbersome and time consuming.

● **Long running tests.** In 9/15 cases, tests and experiments took so long that reviewers did not have time to completely test the artifact. No artifact had instructions to execute long running tests (hours to days long) in a cluster or cloud environment to free up resources on the reviewer's local machine.

● **Issues compiling or running.** In 3/8 cases, reviewers encountered compilation errors. In 6/8 cases, reviewers encountered errors when they executed experiments and tests. In 4/8 cases, reviewers encountered artifact-specific errors that were hard to debug without a deeper understanding of the artifact and its dependencies.

● **Ignored errors.** In one case, two tests failed but the artifact still emitted results that matched its paper. Reviewers then were unsure how to interpret the test failures and eventually assumed that the artifact was functional.

● **Downloads during execution.** In both cases, resources were downloaded over the internet. One reviewer noted that the artifact could stop working after a few years as online resources are deleted. In one case, resources were downloaded whenever a reviewer executed a test case. This made re-executing test cases with different parameters a time consuming process, especially exacerbated by a slow internet connection.

Grid 2: Context for types of positive comments

● **Self-contained.** Two reviewers enthusiastically thanked authors for providing a VM. In one case, the reviewer was using a different operating system than the artifact required, and in another case, a reviewer appreciated having access to an integrated GUI in the VM. Two reviewers commented on how quick and easy it was to test artifacts that were encapsulated in Linux containers. One reviewer was surprised that an artifact was "completely self-contained and running flawlessly the first time!"

● **Lightweight.** One reviewer thanked the artifact authors for providing a separate link to just the artifact's code instead of only providing a VM or container image. This allowed the reviewer to easily inspect the artifact locally. In several cases, reviewers asked artifact authors to remove unnecessary files or to convert Linux VM's to a lighter weight format like a Linux container.

● **Comprehensive documentation.** One reviewer praised the artifact's authors because the "instructions were clear, easy to follow, and explored all aspects of the work." In one exceptional case, artifact authors even documented expected errors. In a different case, reviewers appreciated having a mapping of artifact code to claims made in the paper.

data set can be downloaded in full if needed, and it can be served efficiently over a network filesystem for testing. The two parts, the data package and the code package, would then constitute the artifact.

Therefore a **complete snapshot** of an artifact helps to address ● Issues with dependencies, ● Not enough resources, ● Downloads during execution, ● Long running tests, and ● Self-contained.

Second, an artifact that is provably tested, for example with output from a continuous integration system as proof, is less likely to produce errors for reviewers. This is partly because setup and execution errors are more obvious to authors when artifacts are executed in a different environment than they were developed in. But it also forces authors to test helper scripts, which sometimes are hastily created. An added bonus is that if logs from these proofs are given to reviewers, reviewers can check if a spurious error is cause for concern.

Therefore a **provably tested** artifact helps to address ● Works in limited environments, ● Errors in scripts, ● Issues compiling or running, and ● Ignored errors.

Third, an artifact that is structured in a standard way is easier for reviewers to understand and use. For example, an artifact that has compilation scripts, execution scripts, data files, and documentation in predictable places can have simpler, less error-prone documentation. While it is not beneficial to artifact authors to over-standardize, at minimum, many commands needed to setup and test an artifact could be encapsulated in testable scripts and stored in predictable places.

Therefore a **consistently structured** artifact helps to address ● Problems with docs, ● Too complicated, and ● Comprehensive documentation.

While these themes — **complete snapshot**, **provably tested**, and **consistently structured** — cover many of the topics in Parts 3-4, there are important issues that they don't address.

First, not all artifact execution tools (e.g. virtual machine managers, container runtimes) work well on every operating system and on every physical machine. Furthermore, all possible setups are unreasonably hard for authors to test. It is therefore important for artifacts to be formatted using an open standard so reviewers can use the best, most freely available tools. Proprietary formats tend to offer users less choice so it would behoove authors to avoid them. In this study, it was encouraging to see that 93% of the artifacts we observed could be run on an open source operating system (Linux) and could be encapsulated using an open image format (OVF [7] or OCI [8]).

To summarize the best practices we have highlighted so far, artifacts are easier to setup, execute, and review if:

- They are a **complete snapshot** of code, dependencies, and data
- They are **provably tested**
- They are **consistently structured**
- Their environment and format are as **transparent** and **open** as possible

While the first three properties may be to a large degree achieved by constantly improving guidelines and tools, the last two depend on the cooperative efforts of software engineering communities. In the next section we discuss ongoing work to help authors achieve the first three.

6. Ongoing and Future Work

Inspired by the the data collected in this study and "war stories" from researchers, we are experimenting with ways to encapsulate research demos in Linux containers so they are easy to test and change. A demo is a complete snapshot of a research prototype, containing algorithmic code, software dependencies, and often a small example data set. The demos can be executed by a continuous integration system to provably test their functionality over time, and they will contain auto-generated documentation from simple metadata. The main goal of this work is to make research artifacts easier to use by artifact evaluation committees, by researchers in academic institutions, and by the public at large [11]. A side effect of this work is that artifact authors can use the demo format and tools to easily prepare artifacts. When our first demos are shared, a followup report will also be shared describing the approach in greater detail.

A complementary project called Dockstore encapsulates genomics research code in docker containers [12]. Inspired by the vast amounts of data required to run genomics analyses, the Dockstore project promotes the distributed storage and use of genomics data. Furthermore, it allows compatible encapsulated tools to be linked together, for example, to create more complex systems. While the Dockstore project promotes reusing research artifacts in a flexible dataflow-like model, our work is more focused on making individual artifacts easy to understand and use on their own.

Also worth noting is the ongoing work of the Journal of Open Source Software (JOSS) [9], [10], where research software engineers and the wider scientific community can receive a citable DOI for high quality research software. This allows authors to get citation credit for work that may not be accepted by traditional scientific journals yet still is of high value to the research community. Moreover, JOSS has a peer-review process where submitted artifacts are held to engineering level standards of documentation and testing.

Finally, future work ought to address the preservation of research artifacts so that accessible, peer-reviewed artifacts can be used for many years to come. Organizations like the Software Sustainability Institute [13] and projects like Software Heritage [14], among others, might collaborate in preserving not only source code but also peer-reviewed artifact images, dramatically increasing the chance that an artifact can be setup and used again.

This effort could be more manageable if artifacts met certain constraints, for example:

- The artifact was peer-reviewed, for example by JOSS or by an artifact evaluation committee
- The artifact earned, at minimum, the equivalent of the ACM's "available" and "functional" badges

- Its corresponding research paper was published in an open access journal

These constraints would reward fully open, peer-reviewed research from paper to artifact. While it is an ambitious project, we argue that preserving knowledge as important as creating it.

ACKNOWLEDGMENT

We thank Jan Vitek and Colin Gordon for including us as a member of the OOPSLA 2019 artifact evaluation committee, and we thank the authors of the 28 papers we were able to observe, as without their permission we could not have done this work.

LICENSE

REFERENCES

[1] Baker, M. 1,500 scientists lift the lid on reproducibility. Nature. 2016.

[2] N. Ferro, N. Fuhr, K. Jrvelin, N. Kando, M. Lippold, and J. Zobel. Increasing Reproducibility in IR: Findings from the Dagstuhl Seminar on Reproducibility of Data-Oriented Experiments in eScience. SIGIR Forum. June 2016.

[3] Hettrick S, Antonioletti M, Carr L, Chue Hong N, Crouch S, De Roure D, Emsley I,Goble C, Hay A, Inupakutika D, Jackson M, Nenadic A, Parkinson T, Parsons MI,Pawlik A, Peru G, Proeme A, Robinson J, Sufi S. 2014. UK research software survey 2014. https://datashare.is.ed.ac.uk/handle/10283/785. May 2015.

[4] M. R. Munaf, BB. A. Nosek, D. V. M. Bishop, K. S. Button, C. D. Chambers, N. Percie du Sert, U. Simonsohn, E.-J. Wagenmakers, J. J. Ware, and J. P. A. Ioannidis. A manifesto for reproducible science. Nature Human Behaviour. January 2017.

[5] ACM Artifact Review and Badging. https://www.acm.org/publications/policies/artifact-review-badging. Accessed August 2019.

[6] OOPSLA: Object-oriented Programming, Systems, Languages, and Applications. https://2019.splashcon.org/track/splash-2019-oopsla. Accessed August 2019.

[7] Open Virtualization Format. https://www.dmtf.org/standards/ovf. Accessed August 2019.

[8] Open Container Initiative. https://www.opencontainers.org. Accessed August 2019.

[9] Smith AM, Niemeyer KE, Katz DS, Barba LA, Githinji G, Gymrek M, Huff KD, Madan CR, Cabunoc Mayes A, Moerman KM, Prins P, Ram K, Rokem A, Teal TK, Valls Guimera R, Vanderplas JT. Journal of Open Source Software (JOSS): design and first-year review. PeerJ Computer Science. 2018.

[10] JOSS: Journal of Open Source Software. https://joss.theoj.org. Accessed August 2019.

[11] Public Code. https://www.public-code.org. Accessed August 2019.

[12] O'Connor BD, Yuen D, Chung V et al. The Dockstore: enabling modular, community-focused sharing of Docker-based genomics tools and workflows [version 1; peer-review: 2 approved]. F1000Research. Jaunuary 2017.

[13] Crouch, Stephen; Chue Hong, Neil; Hettrick, Simon; Jackson, Mike; Pawlik, Aleksandra; Sufi, Shoaib; Carr, Les; De Roure, David; Goble, Carole; Parsons, Mark, "The Software Sustainability Institute: Changing Research Software Attitudes and Practices," Computing in Science Engineering. 2013.

[14] Jean-Franois Abramatic, Roberto Di Cosmo, Stefano Zacchiroli, Building the Universal Archive of Source Code, Communications of the ACM. 2018.