Longevity of Artifacts in Leading Parallel and Distributed Systems Conferences: a Review of the State of the Practice in 2023

Quentin Guilloteau, Florina M Ciorba, Millian Poquet, Dorian Goepp, Olivier Richard

Longevity of Artifacts in Leading Parallel and Distributed Systems Conferences: a Review of the State of the Practice in 2023

Quentin Guilloteau
Florina M. Ciorba
Quentin.Guilloteau@unibas.ch
Florina.Ciorba@unibas.ch
University of Basel
Basel, Switzerland

Millian Poquet
Millian.Poquet@irit.fr
Univ. Toulouse, CNRS, IRIT
Toulouse, France

Dorian Goepp
Olivier Richard
Dorian.Goepp@inria.fr
Olivier.Richard@inria.fr
Univ. Grenoble Alpes, Inria, CNRS, LIG
Grenoble, France

ABSTRACT

Reproducibility is the cornerstone of science. Many scientific communities have been struck by the reproducibility crisis, and computer science is no exception. Its answer has been to require artifact evaluations along with accepted articles and award badges to reward authors for their efforts to support 'reproducibility.' Authors voluntarily submit artifacts associated with a submission to reviewers who decide their 'reproducibility' properties. We argue that the notion of 'reproducibility' considered by such badges is limited and misses important aspects of the reproducibility crisis. In this article, we survey almost 300 articles from five leading conferences on parallel and distributed systems held in 2023 (CCGrid, EuroSys, OSDI, PPoPP, and SC). For each article, we gather information about its artifacts (how it was shared, under which experimental setup, and how the software environment was generated and shared), as well as the reproducibility badges awarded. By reviewing the methods and tools used to create and share artifacts in a technical, in-depth, and article content-agnostic manner, we found that the state of practice does not address reproducibility in terms of artifact *longevity* and we expose eight observations that support this finding. To address the longevity of artifacts, we propose a new badge based on source code, experimental setup, and software environment. These criteria will allow rewarding artifacts expected to withstand the test of time. This work aims to shed light on the issue of long-term reproducibility in parallel and distributed systems and to start a discussion in the community towards addressing the issue.

CCS CONCEPTS

• **General and reference → Empirical studies**.

KEYWORDS

Reproducibility, Artifact Evaluation, Badges, Longevity

ACM Reference Format:
Quentin Guilloteau, Florina M. Ciorba, Millian Poquet, Dorian Goepp, and Olivier Richard. 2024. Longevity of Artifacts in Leading Parallel and Distributed Systems Conferences: a Review of the State of the Practice in 2023. In *Proceedings of 2024 ACM Conference on Reproducibility and Replicability (ACM REP'24).* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

ACM REP'24, June 18-20, 2024, Rennes, France

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The scientific community as a whole is traversing a reproducibility crisis for the last decade. Computer science is not an exception to this crisis [66, 4]. The reproducibility of research is essential to build solid knowledge and increase reliability and confidence in the results, while limiting the methodology and analysis bias. In 2015, Collberg et al. [15] studied the reproducibility of 402 experimental articles published in *system* conferences and journals of 2011 and 2012. Each of the articles studied linked the source code used to perform their experiments. Of the 402 articles, 46% were not reproducible. The main causes were: *(i)* the source code was not available, *(ii)* the code did not compile or run, *(iii)* the experiments required specific hardware.

To reward authors of reproducible articles, several publishers, such as ACM or Springer, set up a peer review-based artifact evaluation for each submission. This peer review process of the experimental artifact can award one or several badges to the authors based on the level of reproducibility of their artifacts.

The term reproducibility is often used in a broad sense and gathers several concepts. ACM proposed definitions for the reproducibility terminology, which are used to validate the artifacts submitted [1]. These definitions themselves are based on the International Vocabulary of Metrology [8], and comprise three levels: (i) *Repeatable*: the measurements can be obtained again by the people at the origin of the work. (ii) *Reproducible*: the measurements can be obtained again by people who do not belong to the original work team but have the original artifact of the authors. (iii) *Replicable*: the measurements can be obtained again by people who do not belong to the original work team but with artifacts they developed independently.

Reproducibility is harder to achieve when the artifacts of an experiment do not include the software environment in which it was conducted. Indeed, side effects due to the environment can occur and change the results of the experiment. It is easy to forget to include an element in the software environment that has an impact on the performance of the experiment. For instance, performance but also the result of a simple C application may depend on the compilation options [76] or also from the quantity of UNIX environment variables [57].

Most of the current solutions in terms of 'reproducibility' involve storing artifacts (system images, containers, virtual machines) and

the replay of experiments [69, 10, 11]. Even if this is an important step towards reproducibility, there is no guarantee that the software environment can be re-built in the future, and thus no guarantee that the experiments can be re-run if the artifacts disappear.

Evaluation of artifacts from conference papers is typically conducted soon after their initial construction. Thus, it is highly likely that the construction of the artifacts uses package mirrors (`apt`, `rpm`, etc.) in a state or version similar to that of the submitted artifacts.

This raises the question of: What will happen when someone tries to rebuild the artifact environment 1, 5, or 10 years into the future? The objective of science is to be based on robust work to advance the frontiers of knowledge (*Stand on the shoulders of giants* [58]). Such '**short-term reproducibility**' is a major obstacle to scientific progress and is in complete opposition to Open Science [82]. No one would expect a mathematical proof to change over time or even completely disappear. We believe that artifact description currently mainly targets artifact reviewers, **but, more importantly, it should target future readers and researchers.**

We believe that the concept that should be highlighted here is **variation** [55, 31]. This means allowing a third party to use the environment defined for an experiment to investigate the same idea or another research idea. An example of variation would be to change the MPI implementation used in an experiment (*e.g.,* MPICH instead of OpenMPI). Being able to introduce such a variation requires the initial environment to be correctly defined.

However, even if variation is the end goal, we claim that the current state of practice in Artifact Description (AD) and Evaluation (AE) does not yet fulfill the main reproducibility properties, in particular in terms of the reproducibility of the experimental software environment.

This article analyzes the *longevity* of existing ADs of the surveyed articles and exposes seven observations summarizing our findings. The *key novelty* of this article is to propose a badge that complements existing reproducibility badges and supports the longevity of artifacts to withstand the test of time.

This article is structured as follows. Section 2 presents the context and work related to artifact evaluation. The objectives and methodology of this study are presented in Section 3. In Section 4, we review almost 300 articles from five leading parallel and distributed systems conferences (CCGrid, EuroSys, OSDI, PPoPP, and SC) of 2023, and discuss the state of the practice of artifact sharing. Based on the findings described in Section 4, in Section 5 we propose a new badge that accounts for artifact *longevity* and discuss the limitations of this study in Section 6. Finally, Section 7 summarizes the work and presents final remarks and perspectives.

2 BACKGROUND AND RELATED WORK

The reproducibility-related definitions proposed by ACM [1] carry some confusion, and the community did not reach a clear consensus [65, 6]. In this paper, we modify the definition from [70] to add the dimension of the experimental platform: "An experiment is *reproducible* if the source code, the raw data, the analysis scripts are available, their usage sufficiently described for someone to reproduce the experiments and analysis, and the access to the experimental platform used is open". An *artifact* is a result of self-contained work with a context-specific purpose [54].

In 2015, Hunold [50] conducted a survey among participants at the Euro-Par conference to assess the vision of the parallel computing community on reproducibility questions. When asked about the main reasons for not making the source code/raw data/data processing available, the participants answered that: "*it is irrelevant because evolution is too fast*" (90%), "*it is not rewarding*" (87%), "*I want to retain a competitive advantage*" (84%). The second most popular answer is quite interesting, since the AE processes were not very popular at the time of the survey (the very first was in 2011 at the ESC/FSE conference). Since then, the sharing of artifacts and their evaluation has become an established and accepted practice with benefits for the community [46]. The work by publishers with the badging system aimed to reward authors for sharing their artifacts. It would be interesting to conduct the survey again today to quantify the impact of AE on this question, as we believe that reproducibility and its challenges have since gained greater visibility. Badges have been shown to be an effective strategy to incentivize authors to make their research data available [52, 71], but have not yet shown a significant impact on the visibility of the articles [84, 33, 48].

In [47], the authors surveyed the members of the artifact evaluation committees of computer science conferences about their expectations of artifacts and the artifact review process. They found that despite the call for artifacts that expressed expected observable qualities of the submitted artifacts, there was no consensus on what the expected qualities should be. The authors of [12] proposed a global "quality indicator" for research artifacts with a detailed framework, but it is not focused on reproducibility and does not integrate with the current badge system. This lack of consensus leaves reviewers without guidelines for correctly and uniformly evaluating artifacts, which has been shown to be frustrating for reviewers [7]. Furthermore, the study showed that there is a lack of reviewer experience for reviewing artifacts.

Reviewing and reusing artifacts require two different points of view. Reviewing focuses more on the overall "quality" of the artifacts (*i.e.,* completeness, documentation), while the readers are more interested in their reusability. An answer in the survey conducted by Hermann et al. [47] explains that the experiments presented in an article should be reproducible and that good documentation and ease of setup are only bonuses.

Reusing artifacts poses problems when used in comparison with other methods. When researchers want to compare their new method with a method from the state-of-the-art, they either need to reimplement the method from scratch if no artifact is available, or, if the artifact is available, researchers need to adapt the code of the artifact to enable a comparison between the methods. In both cases, it is not the original work that is being compared but a modified version of it, which might lead to different results. An artifact with all reproducibility badges might not be reusable and comparable "as is" by other researchers. A solution that should be promoted by committees is the implementation of the authors' solutions on collaborative benchmarking frameworks. Some examples of such collaborative frameworks exist for optimization problems [56], Edge-to-Cloud experiments [68], or networking experiments [73].

The survey participants [47] also stated that the most important thing is the availability of artifacts, rather than its reproducibility.

The last decade has seen the creation of independent online scientific journals to reward software and reproducibility. The most popular example is probably the *Journal of Open Source Software* (JOSS) [75] that publishes articles about open source *research software*. The review process, openly accessible as GitHub issues, includes a thorough inspection of a submission's source code, the documentation of the software, and a run-through of some examples. In the field of Image Processing, the online journal *Image Processing On Line* (IPOL) [16] requires the authors to implement the algorithms proposed in their article and to make the implementation available through an online demonstration for readers to explore and play with. This requirement forces the authors to share their code along with their article. IPOL noted that this requirement also helped authors to improve their algorithms, as actually implementing the algorithms might raise some undetected edge cases. As the review process for journals is often much longer than for conferences, reviewers have more time to investigate the artifacts and iterate with the authors ways to improve the artifacts. In 2021, the journal on *Transactions on Parallel and Distributed Systems* (TPDS) started a program where researchers can submit short "*critique*" articles that present their experiences in reproducing published results, and that will be linked to the original publication if accepted [80].

Studies on the artifact process focus mainly on high-level characteristics, as well as on the availability and citations of artifacts [52, 71, 84, 33, 48]. In this work, we propose a technical, in-depth, and article content-agnostic review of the methods and tools used to create and share artifacts.

3 METHODOLOGY FOR EVALUATING ARTIFACT SHARING

In this section, we survey 296 articles from 5 of the leading *parallel and distributed systems* conferences of 2023, namely, CCGrid, EuroSys, OSDI, PPoPP, and SC. These conferences used an Artifact Description/Artifact Evaluation (AD/AE) process for the *accepted* articles. This AD/AE process usually consists of the authors writing an AD as an appendix of the article to show how to get and use the artifact, how to install the dependencies, what the different experiments are, and their estimated duration [63, 23, 49]. The AD section is typically one or two pages long (in a double column layout). This AD is complemented in practice by a web link provided by the authors to a more detailed description of the artifact.

We selected five of the leading parallel and distributed systems conferences held in 2023 that had a AD process and examined their published proceedings. We formulated the questions below to guide the survey of all articles in the proceedings. For each article, we note the answer to these questions:

(1) How many reproducibility badges were awarded and which badges were awarded to the article?
(2) Does the article have an AD section?
(3) Whether the article shared the URL of the artifact (it does not have to be in the AD), and whether the URL is still valid?
(4) How was the source code shared: `git` repository (*e.g.,* GitHub, GitLab), Zenodo, or a combination of solutions?
(5) If the source code has been shared via a `git` repository, we record the number of commits and check whether a precise commit was specified by the authors.

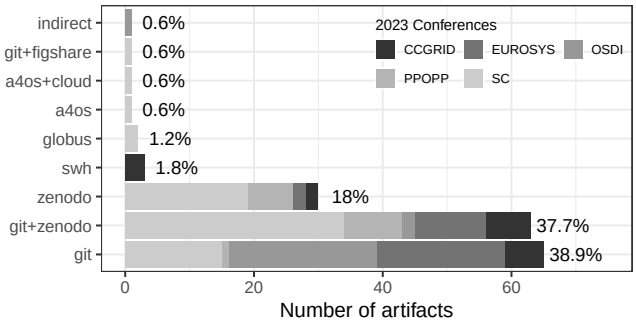


Figure 1: Methods used by the authors to share artifacts. The analysis characterizes the 154 artifacts available (Table 1). The state-of-the-practice is dominated by `git` URLs, Zenodo archives, and a combination of the two (`git+zenodo`).

(6) How were the experiments performed (*e.g.,* local machines, shared test-beds, proprietary machines, supercomputers, simulation)?
(7) How was the software environment described and shared?
(8) What was the workflow of the experiments? (initially not part of the survey questions)

In the following, we study five aspects of the ADs: artifact badges and availability in Section 4.1 (points 1, 2, 3 above), source code availability in Section 4.2 (points 4, 5 above), experimental platform used in Section 4.3 (point 6 above), description and sharing of the software environment in Section 4.4 (point 7 above), and workflow of the experiments in Section 4.5 (point 8 above). Note that the initial survey was not designed to record the workflow used in the experiments and that this question arose during the analysis of the artifacts.

4 RESULTS

4.1 Artifact badges and availability

Table 1 summarizes our evaluations, including the number of articles for each conference and how many of them have AD. The first surprising observation is that only about 20% of the articles with the "Artifact available" badge also have an AD section in the corresponding conference proceedings version. We presume that the authors either forgot or declined to include this section in the final version of the article. This is an unfortunate finding, as we believe that the **AD is valuable both for the artifact reviewer and for future readers of the article.**

Observation 1: Artifact badges and availability

Not all the papers rewarded with the "Artifact available" badge have an artifact description in the proceeding version.

4.2 Source code availability

Figure 1 shows how the artifacts were shared in the articles. Note that several articles shared a link to their artifacts without having

Table 1: Papers by considered conferences. The CORE rank [17] is in between parenthesis.

	Papers			Badges			Artifact URL	
Conference	Accepted	Found PDF	Artifact Section	Available	Functional	Reproduced	Specified	Available
CCGRID (A)	58	58	18	20	0	17	18	17
EUROSYS (A)	54	54	27	31	24	8	33	29
OSDI (A*)	55	55	13	28	30	26	26	25
PPOPP (A)	31	31	18	22	21	17	17	17
SC (A)	98	98	81	60	46	33	74	66
Total	296	296	157	161	121	101	168	154

an AD section or a badge. Most articles simply include the URL of their `git` repository. Certain articles shared their code with Zenodo [87] or Figshare [32], while others shared both with a `git` URL and Zenodo. A minority of articles used Software-Heritage [45] (abbreviated `swh` in Figure 1), Globus [37], or personal `cloud` drives.

Several authors used `anonymous.4open.science` [78] (abbreviated `a4os` in Figure 1) that allows users to share an anonymous copy of a public repository on GitHub with reviewers. This is particularly useful for double-blind reviews. However, for all articles surveyed, all links to this service were dead and there was no way to retrieve the original `git` repository. We believe that the links simply expired, which gives rise to issues of reproducibility by future researchers. Furthermore, as long as the AE process does not count in the accept/reject decision, having a double-blind review for the AE only limits communication between reviewers and authors. If the results of the AE will be taken into account for the decision, then the community may need to investigate ways to perform the AE in a double-blind manner. An easy solution would be to use tools such as `anonymous.4open.science` [78] for the review and then replace the URL with a (more) persistent URL in the camera-ready version of the article. Some communities use third parties to anonymously review artifacts, especially when they contain sensitive data [64].

A minority of authors shared their artifacts through an indirect link. In most cases, this link points to the author's personal Web page, where there is the true link to access the artifact. The drawback of this approach is that if the author's Web page is no longer accessible, the link given in the article is no longer valid. Similar comments apply to the sharing of artifacts with a link to a personal cloud space (*e.g.,* Google Drive).

Sharing only with a `git` URL can lead to traceability issues. For instance, only 6% of the articles that shared artifacts via a `git` URL mentioned the commit used for the experiments. Such a solution could be satisfactory for the AE since the delay between the submission of the article and the evaluation of its artifact is short enough for the source code to be unaltered or in a similar state. However, for future researchers aiming to build upon these artifacts, it is nearly impossible to know which version of the code was used. Another drawback of only using a `git` URL is that the source code hosted on forges (*e.g.,* GitHub, GitLab) might not be available forever. For instance, authors could decide to delete or rename their repository, invalidating the URL given in the article. A better solution would be to use an institutional account on the forges to store the `git` repository. However, in the worst case, the entire source forge may need to close, making all repositories unavailable (*e.g.,* Google Code [38], GForge Inria).

One solution proposed by the conferences' reproducibility guidelines is to archive the code via Zenodo or Figshare, and then refer to the DOI generated by these archive websites in the AD section. This has the advantage of giving a snapshot of the source code as it was at the time of submission and allows future use of the code. However, storing the source code on Zenodo has a simple drawback: There is no possibility of partial code exploration. From the point of view of future researchers, having to download potentially large Zenodo archives to explore a few source files may be cumbersome and may hinder engagement with the source code in the artifact. Archiving can also break the link between the original `git` URL if not archived correctly. Zenodo integrates with GitHub [36], allowing to archive *releases* of a repository. This is why certain authors share the `git` URL and a Zenodo archive. If the link between the repository and the Zenodo archive breaks (*e.g.,* `git` repository becomes unavailable), future researchers are left with a single commit of the source code, and all the history of the project, which contributes to the understanding and extensibility of the project, is lost. Several artifacts shared through Zenodo are actually archives of a `git` repository and include the `.git` folder, and thus the history of the project. Zenodo and Figshare are adapted to archive datasets and binaries, not to the source code. Zenodo and Figshare are heavily used because of the requirements from the artifact review committee to have well-identified and citable software, which goes through giving a DOI to the artifact. However, these solutions are more appropriate for raw data and binaries, not for source code [2, 24].

A more appropriate solution is to use Software-Heritage [45, 25]. Similarly to Zenodo, it offers permanent storage of source code, with the same interface as usual source forges (*e.g.,* GitHub, GitLab, etc.). This means that future researchers can explore the source code through an intuitive web interface without having to download any archive. Software-Heritage also refers to the original source, so that future researchers can access it if still available. For example, [43] is the Software-Heritage archive of this article repository.

During this survey, we observed a surprising low number of commits to the repositories linked in the articles when shared with `git` (*i.e.,* `git`, `git+zenodo`, `git+figshare` on Figure 1). We discovered that 25% of the repositories have no more than 6 commits and that 50% of the repositories have less than 20 commits. These repositories appear to be a "dump" of the source code with some extra

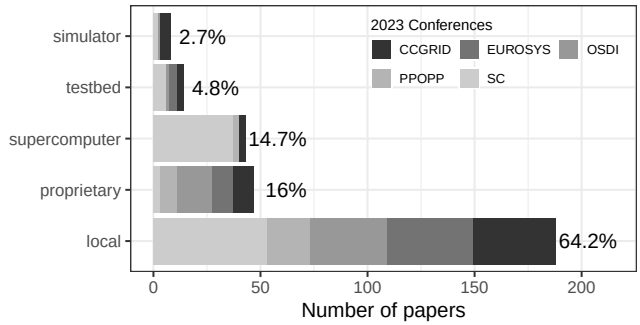


Figure 2: Experimental platform used in the surveyed articles. Most authors use the local machines at their disposal. Certain authors also use supercomputers to experiment on state-of-the-art systems. More concerning is the number of articles relying on proprietary platforms, such as Amazon Web Services, Google Cloud, Microsoft Azure. Finally, a small proportion of the articles uses shared testbeds.

commits for documentation. Such practices do not allow reviewers and future researchers to explore the "true" history of the project, which is contrary to the traceability principle of Open Science [82]. The observations made for a standalone Zenodo archive also apply here. It also casts doubt on the authors' good practices in terms of the traceability of the experiments. We believe that for certain authors, the AE process and reproducibility may only be a secondary consideration.

Observation 2: Sharing source code

The practice of sharing code through a `git` URL might result in future code unavailability. Archiving via Zenodo is a better alternative, but may introduce friction for future exploration. Using Software-Heritage appears to be the best available solution to permanently share source code.

4.3 Experimental platforms

A sufficiently detailed description of the hardware used for any experiment that exhibits a particular behavior or performance evaluation is crucial for reproducibility.

Figure 2 shows the types of platforms on which the experiments were performed for *all* the surveyed articles, with or without AD or a badge. Note that some artefacts used several platforms. Most of the experimental platforms were local machines (`local`), but the description of the machine (*i.e.,* CPU, GPU, disk, etc.) is given. This still makes it difficult for reviewers and future researchers to find the exact hardware or something closest to the hardware used. In some ADs, we observed that the authors provided access to their local machines by giving the IP address and the password to connect.

A better solution would be to use open and shared platforms, also called *testbeds* [60]. Chameleon [51], Grid'5000 [5], or CloudLab [30] are examples of such testbeds. Testbeds are not frequently used, being used only in about 5% of the articles. In practice, proprietary platforms such as Amazon Web Services, Microsoft Azure, Google Cloud, are used more frequently, in 16% of the articles. Even if this allows reviewers to more easily access probably similar machines in the short term, it locks the experiments, and thus their reproducibility, behind a paywall. This goes against the Open Science principles [82]. Using proprietary platforms also raises the question of who should pay to reproduce the results of the authors. Some authors using such platforms wrote in their AD the estimated monetary cost of rerunning the experiments.

Similarly, several articles (in particular those from the Supercomputing conference (`SC`)) used supercomputers to conduct experiments. While supercomputers are at the bleeding edge of technology, having access to such a system is restrictive and can take several weeks or months before obtaining access.

Observation 3: Experimental platform

Most articles use machines that are difficult to access (local, supercomputer, or proprietary). Testbeds are underrepresented in the state of the practice, but appear to be better suited for reproducibility [60].

4.4 Software environment

After downloading the correct version of the code on the correct platform, reviewers must configure the correct software environment to execute the experiments. Figure 3 shows the different techniques used to describe and share the software environment in ADs. Note that an AD may use *several* of these techniques.

In the following, we go through the methods observed to share the artifact software environment and discuss their reproducibility.

4.4.1 Images. Figure 4a shows the tools used to capture the software environment of the experiments. Contrary to predictions made in 2017 [74], most AD do not report using any particular tool. However, certain AD report using virtualization tools, such as containers or virtual machines.

The entire software stack is typically encapsulated in an *image*. This image can then be deployed on machines to conduct the experiments. A way to generate a system image is to start from a base image, deploy this image, execute the commands required to set up the desired environment, and finally compress the image. Platforms such as *Grid'5000* [5] and Chameleon [51] offer such tools to their users (respectively `tgz-g5k` [40] and `cc-snapshot` [14]). In the context of repeatability and replicability, if the image remains available, this method of producing system images is adequate at best. Concerning the traceability of the build, one cannot verify the commands that have been used to generate the image, and thus one relies entirely on the documentation from the experimenter. Moreover, such images are not suitable to be versioned with tools such as `git` as they are in binary format. In case the image is no longer available, re-building the exact image may be complex.

Figure 4b shows the availability state of the images for AE. We observe that most authors who use an image make it available in a binary cache such as DockerHub. However, DockerHub does not offer permanent image storage. Another solution is to archive the

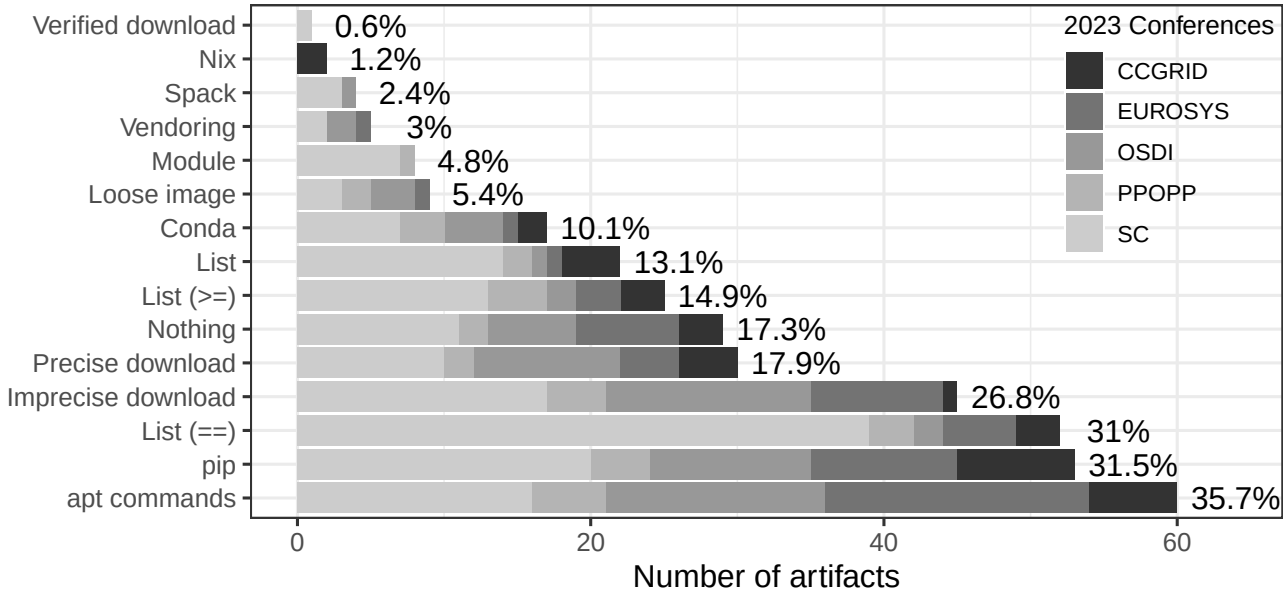
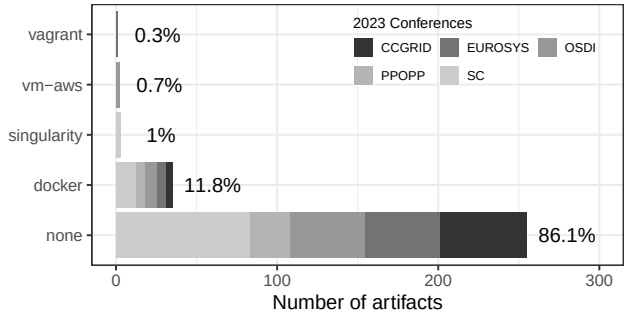
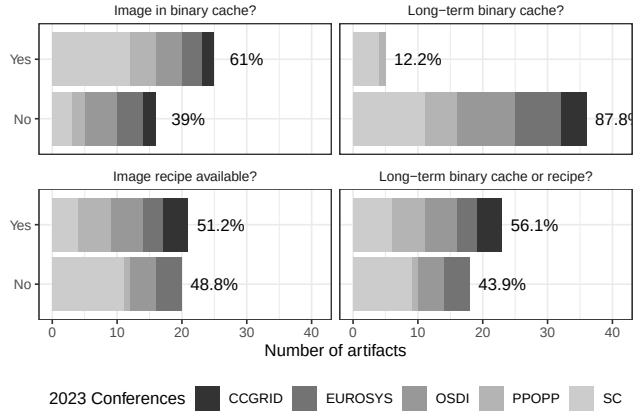


Figure 3: Techniques used to share the software environment in the ADs. An artifact can use several techniques.



(a) Most artifacts used no tool or technology to generate/package their software environment. Others used virtualization tools (*e.g.,* containers or virtual machines), the most frequent being Docker.



(b) Questions related to the storing the prebuilt images and their recipes for artifacts using virtualization tools.

Figure 4: Tools and technologies used to generate and package the software environment for the AE (Figure 4a), and the state of the image and its recipe in the case of virtual tools (Figure 4b).

image in a long-term binary cache, such as Zenodo. However, this is done rarely by the authors, only in 12% of the papers.

A better approach to (re)generate and share images is to use *recipes*. Recipes, such as `Dockerfiles` for Docker containers or Kameleon recipes [72] for system images, are a sequence of commands to execute on a base image to generate the desired environment. The text format of the recipes makes them more suitable for version control, sharing, and reconstructing. Base images often have several versions, which are identified by labels called *tags*. In the case of Docker, the tag for the latest version is often called '`latest`'. Basing an environment on this tag breaks traceability and, thus, the reconstruction of the image itself. Indeed, if a newer version is available at the time of a future rebuild of the environment, then the image will be based on this newer version and not the original version. Another important question is to know whether the base image and all the versions can themselves be re-built, and if it is not the case, what is the permanence of the platforms hosting those images? For instance, the longevity of the `Nvidia/CUDA` Docker image is only 6 months; after 6 months, the Nvidia administrators of DockerHub delete the images [61]. However, in Figure 4b we see

that less than half of the ADs that use an image do not share the recipe, or we were unable to find the recipe to inspect or rebuild the image. This means that if the image is not in a binary cache, then it is impossible to rebuild it exactly. The row `Loose image` in Figure 3 shows the articles that based their software environments on a short *longevity* image.

4.4.2 List of package versions. One of the popular approaches to share the software environment is simply to list the dependencies of the artifact. We observed several levels of this listing approach. The first level is to only list the name of the dependencies (`List` in Figure 3). In this case, reviewers or future researchers do not have information about the versions used or whether there is any required feature from the dependencies. Future versions of a dependency might have introduced breaking changes that make the artifact unusable. The authors can then give a minimum version to use (`List (>=)` in Figure 3), for example `gcc >= 10.0.0`. Although this gives at least a lower bound on the versions, it does not offer a guarantee that any future version will also work. Finally, the most popular approach is to give the version used for each and all dependencies (`List (==)` in Figure 3). Listing all dependencies by hand raises several important questions. Are actually *all* the dependencies listed? What about the dependencies of the dependencies? How can we bring a system in the same state as the original system? An answer to these questions is provided at the end of Section 4.4.

Observation 4: Listing dependencies

Simply listing the software dependencies and their version is not enough to regenerate the correct software environment.

4.4.3 Package managers' installation commands. Another popular way to describe the software environment is to list the installation commands in the package manager (*e.g.,* `apt`, `yum`). These commands typically take the form:

```
sudo apt-get update
sudo apt-get install packageA packageB
```

Another pertinent question arises here: What are the versions of the installed packages? Indeed, calls to `apt-get update` (or equivalent for other package managers) make the software environment depend on the state of the mirror of the package manager at the time the author did the experiments, or on the configuration of the package manager which should thus also be included in the artifact. For AE, the mirror may not change significantly between the time of the experiments and the time of the review. However, the probability that in 5 or 10 years the mirror will be in the same state is very low, and the installed versions will not be exact the same as in the experiments of the authors. This approach also implicitly defines a dependency on the operating system distribution that must be used to recreate the original environment.

There are "workarounds" to make sure that the packages installed via classical package managers are the expected ones. For example, using a *snapshot* of the mirror [21]. These snapshots are a dump of the mirror at a given time and users can then install packages from these snapshots using the usual interface of the package manager. However, the use of snapshots can cause issues. In particular, what if the package installed from the snapshot creates a conflict with a package already installed on the system? This is especially the case for systems based on the Filesystem Hierarchy Standard (FHS), such as Debian-based distributions, where all binaries and libraries are stored under `/usr/bin` and `/usr/lib`. Also, what happens to the already installed packages if the artifact requires the installation of an old version of the `glibc`? One solution would be to use a virtualization tool such as a container or virtual machine, but, as seen in Section 4.4.1, they have their own reproducibility issues.

Using snapshots makes it more difficult to introduce variation in the software environment. Indeed, installing more recent packages might be tedious or introduce conflicts with the installed packages.

Observation 5: Classical package managers

Installing dependencies through classical package managers (*e.g.,* `apt`, `yum`) creates a dependency on an uncontrollable state: the state of the mirror of the package manager. Freezing the state of the mirror introduces new compatibility problems with the underlying system and hinders the introduction of variation.

4.4.4 `pip` *and* `conda`. When the software environment contains only Python packages, freezing the dependencies with `pip` (`pip freeze`) is not enough. `pip` only describes the Python environment, and ignores the system dependencies that numerous packages have. For example, freezing an environment containing `zmq` Python package will not freeze the ZeroMQ system package installed on the system. Even if re-creating a Python environment from a `requirements.txt` is simple, installing a list of system packages with specific version is, on the other hand, much more complex. In the best case, the repository includes a `requirements.txt` that lists all Python dependencies with the *exact* versions. However, in practice, we observed the same issues as presented in Section 4.4.2 when the authors provided a list of dependencies without version or with a loose version.

4.4.5 Downloading from the outside world. A common practice when authors need to install a dependency unavailable through classical package managers is to install it from source. For this, the authors indicate in the AD how to download the dependency and how to build it. However, when cloning a `git` repository, a common error is not specifying the commit to use. If no commit is specified, `git` will by default use the latest commit of the main branch, which could be completely different between the moment of artifact review and 10 years into the future. The same goes for archives downloaded via `wget`/`curl`, typically in the form

```
curl https://website.com/download/release-latest.tar.gz
```

the outcome of which varies with time. Both of these approaches were labeled `Imprecise download` in Figure 3.

Moreover, the downloaded `git` repository could disappear in the future, and therefore cloning from Software-Heritage would be more robust than cloning from a forge (*e.g.,*GitHub, GitLab). The same remark applies to downloaded archives from websites.

Another important point is to check that the downloaded object is indeed the expected one. This can be done by checking the cryptographic hash of the downloaded object and comparing it to

the expected one. Among all articles surveyed, we observed this practice only once (`Verified download` in Figure 3).

Observation 6: Content of downloaded objects

Every object coming from outside the environment must be examined to ensure that it contains the expected content. It is more preferable that the building of the environment fails if the content differs from the expected one, rather than the environment silently building with different content.

4.4.6 Modules. A popular way to manage a software environment in high performance computing (HPC) systems is through *Modules* [34, 13]. Modules allow users to change their environment by "loading" and "unloading" packages and allow one to manage different versions of applications. Under the hood, the modules change the `$PATH` environment variables. One drawback is that loading and unloading modules have side effects on the state of the system and, therefore, might not reset the system to its initial state. Manually loading the correct modules can also be quite error-prone for users. Moreover, modules are system-specific (*e.g.*,compiled MPI with special optimizations for the underlying system). Thus, sharing a module-based environment between two systems might be impossible. As modules are maintained by system administrators, and allow them to limit and control the applications that can be run by users. Moreover, modules do not have infinite longevity and might become unavailable in the future.

4.4.7 Spack. Spack [35] is a package manager similar to `pip` but for system packages and their dependencies. It is possible to export the environment as a text file and rebuild it on another machine. However, the environment produced might not be completely identical. Indeed, Spack uses already present applications on the machine to build packages from the sources. In particular, Spack assumes the presence of a C compiler on the system and will use this C compiler to build the dependencies of the environment. Hence, if two different machines have two different C compilers, then the resulting environment is likely to differ from the desired environment. One clear advantage of Spack is the ease in which one can introduce variation into an environment through the command line. Spack can also be run as an unprivileged user and does not require the approval of the system administrators. Spack will download and build dependencies into a folder located in the user's `$HOME`. However, a drawback of using Spack is that this directory consumes a lot of storage quota and inodes, which are limited in HPC systems.

4.4.8 Vendoring. One way to ensure the use of correct dependencies is to "vendor" them. This means having a copy of the dependencies' source code in the artifact itself and then building the dependencies from source. Authors sometimes use `git submodules` to perform vendoring. However, `submodules` are not copies of dependencies, but simply a link to a specific commit of another `git` repository. Hence, if one of the dependency's repositories disappears, the artifact will not build. Furthermore, vendoring has limits as it cannot reasonably capture *all* dependencies by hand (*e.g.,* C compiler), so it is only limited to "adjacent" dependencies.

4.4.9 Functional package managers. Tools such as Nix [28] or Guix [19] fix most of the problems described in the previous subsections. However, they are only used in about 1% of the artifacts examined. As Nix and Guix share similar concepts, in the following, we will focus on Nix, under the premise that all insights also apply to Guix.

Nix is a pure functional package manager for package reproducibility. A Nix package is defined as a function where the dependencies of the packages represent the inputs of the function, and the body of the function contains the instructions to build the package. Package building is done in a *sandbox* which guarantees to build in a strict and controlled environment. First, the sources are fetched and then their content is verified by Nix. If the hash of the sources differs from the expected hash, Nix stops the build of the package and yields an error. Nix also fetches and builds dependencies recursively. The build commands are then executed in the sandbox with the environment defined by the user. At this stage, no network access or only access to the local file system is possible.

Nix can generate environments that can be assimilated as a multilanguage counterpart to Python's `virtualenv`s. But it can also create containers images (Docker, Singularity, LXC, etc.), virtual machines, or full system images with the operating system NixOS [29]. The process of building an image with classical tools (`Dockerfile`, Kameleon recipe, etc.) is often iterative and arduous. Defining an image with Nix is done in a *declarative* fashion. This has the advantage of making the image build faster when modifying an already built recipe [41]. It also avoids the tedious optimization of the order of operations, which is common when building from a `Dockerfile` [27]. As Nix packages are functions, introducing a variation means changing an argument when the function is called.

Systems like Debian store all packages in the `/usr/bin` and `/usr/lib` directories. This ordering can lead to conflicts between different versions of the same library, and thus limits the introduction of variation in the environment without breaking the system. Unlike FHS-based systems, Nix installs each package in its own directory. These individual directories are stored in the *Nix Store* located at `/nix/store`, in a *read-only* file-system. Each directory name is prefixed with the hash of its sources:

`/nix/store/jqvvkhzkm9irdqdqmb8m1jxahnh2j2yl-nix-2.18.1`

Hence, if a user wants to install a different version of an already installed package, its source code would be different, thus the hash will be different, and Nix will then create a new directory to store the new package. The advantage of this fine-grained isolation method is the *precise* definition of the `$PATH` environment variable to manage software environments.

The definition of packages through functions also facilitates their sharing and distribution. There is a large base of package definitions written by the community and hosted in a `git` repository called `nixpkgs` [59, 67], archived in Software-Heritage. Users can easily base their new packages or environments on those definitions. It is also possible for independent teams and research groups to have their own base of packages. Guix-HPC [44], NUR-Kapack [62], or Ciment-channel [39] are examples of independent package bases for HPC and distributed systems.

Limits of Functional Package Managers. Although tools such as Nix and Guix greatly improve the state of reproducibilityfor software environments, it is still possible to make an impure package

or make it dependent on some exterior state. Nix addresses this issue with the *Flake* experimental feature [81].

To ensure reproducibility and traceability of an environment, Nix requires that all packages and their dependencies have their source code open and that the packages be packaged with Nix. This could seem limiting in the case of proprietary software where the source code is unavailable (Intel compilers, for example). It is still possible to use such proprietary packages with the `impure` mode of Nix, but it breaks the traceability and thus the reproducibility of the software environment.

The construction of packages in a sandbox goes through an isolation mechanism of the file system using `chroot` system call. This feature used to be restricted to users with `root` privileges. In HPC systems, this type of restrictive permission significantly hinders the adoption of Nix or Guix. Thankfully, the *unprivileged user namespace* feature of the Linux Kernel allows users to bypass in most cases this need of specific rights.

As Nix needs to recompile packages that are not available in its binary cache from their source code, it is possible that future rebuilds are impossible if the host of the source code disappears [9]. However, since Software-Heritage now performs frequent archives of open-source repositories, it should be possible to find, when needed, the sources of interest [18].

Finally, these tools also require a change of viewpoint in the way software environments are managed, which might make the learning curve steeper.

Observation 7: Functional package managers

Functional package managers (FPM), *e.g.,* Nix [28] or Guix [19], provide reproducibility guarantees for the software environment produced. FPMs are extremely underused, probably due to their steep learning curve. We believe that such tools are the closest to solving the challenge of reproducibility of software environments.

4.5 Workflow managers

We initially did not plan to record the workflow of the experiments for each article. However, during the survey, we made the striking observation that almost no artifact made use of a workflow manager to conduct experiments. Authors describe the experiments workflow primarily in two ways: using lengthy and fragile `bash` scripts or a `README` file that requires copy-pasting the commands. In certain articles, commands are directly included in the text, making it even harder to read and copy-paste.

As experiments in distributed computing systems can be quite expensive to run (especially if one needs access to a supercomputer or a proprietary system in the cloud), having the possibility to run a subset of the workflow is crucial for reproducibility. For example, a reviewer or future researcher may want to rerun only the data analysis step of the workflow in the artifact (dataset that has been stored on Zenodo, for example) or perhaps add a new combination of parameters to the entire workflow.

Workflow managers [85] have become the standard for executing complex bioinformatic pipelines. However, despite the plethora of available workflow systems [20, 53, 26, 22, 3, 77, 83] the stark observation is that none of the surveyed artifacts used any of them.

Observation 8: Workflow managers

The workflows described in the artifacts are based on manually copy-pasted commands from `README` files or on the execution of fragile `bash` scripts. The community could *greatly* benefit by adopting workflow managers [85].

5 ARTIFACT LONGEVITY BADGE PROPOSAL

We believe that the current badging system misses an important aspect of the quality of artifacts: their *longevity*. By *longevity* we mean the time an artifact will remain in the same state as the state used by the authors. As we have seen in Section 4, the popular tools and methods for sharing source code, the software environment package, or platforms to perform experiments differ in their *longevity* longevity guarantees/quality.

Artifacts that offer *longevity* are much more valuable and impactful to future researchers who may wish to build on them, deserve to be rewarded, and have greater visibility. Table 2 proposes a **first instance** of a grading framework to assess the *longevity* of an artifact based on three criteria: sharing of the source code, the experimental setup used, and the software environment. We propose to grade each aspect on 5 levels ranging from insufficient (Level 0) to best (Level 4). Averaging the score for each criterion yields an overall score for the artifact.

Recommendation: Artifact Longevity Badge

Artifacts that obtain a longevity score of 3 or higher should be awarded the **Artifact Longevity Badge**.

The specifics of the score for each criterion **should further be discussed in the community** beyond the specifics introduced in this work, to reach a consensus on the desired good practices. They are also bound to change as software tools and practices evolve.

Figure 5 shows the distribution of the longevity scores for each of the three criteria and the overall longevity score for the articles surveyed in Section 4. Given the overall longevity scores of the 154 articles surveyed with available artifacts, only 1.2% (or 2 articles) receive the Artifact Longevity Badge. The most penalizing criterion is the software environment, since most articles received a score of 0 because they do not describe their software environment accurately enough. In keeping with the theme of article content-agnostic review, we refrain from identifying in this work the articles that received the newly proposed Artifact Longevity Badge.

6 DISCUSSION

Section 4 exposes the lack of long-term reproducibility of the artifacts. Popular tools and methods do not provide sufficient guarantees concerning the artifacts' *longevity*. By reviewing the state of the practice, we aim to raise awareness about this forgotten yet important dimension of reproducibility. Our proposal of a new badge, in Section 5, to reward authors for creating artifacts with *longevity* aims to **start a community discussion on this topic**.

Table 2: Proposed grading framework for evaluating artifact *longevity*.

Grade	Artifact Longevity Criteria		
[0..4]	Source Code	Experimental Platform	Software environment
0	imprecise version (*e.g.,* `git` repository without *fixed* commit)	not described	not partially *described* (*e.g.,* dependencies list, `apt` commands, imprecise download)
1	fixed version, partial exploration (*e.g.,* `git` with *fixed* commit)	high monetary cost (*e.g.,* Proprietary platforms)	long-term availability, some dependencies (*e.g.,* Vendoring, *precise* download)
2	long-term storage, fixed version (*e.g.,* Archive of a *release*)	difficult access (*e.g.,* Local machines)	shorter-term availability, recipe, most dependencies captured, more precise rebuild (*e.g.,* Spack)
3	long-term storage, fixed version, history (*e.g.,* Archive of a repository with history)	longer-term access, difficult access, low monetary cost (*e.g.,* Supercomputer)	long-term availability, available recipe, all dependencies captured, imprecise rebuild (*e.g.,* Long-term storage of the image and recipe)
4	long-term storage, fixed version, history, partial exploration (*e.g.,* Software-Heritage)	longer-term access, easy access, low to no monetary cost (*e.g.,* Testbeds, simulator)	long-term availability, recipe, exact rebuild, all dependencies captured (*e.g.,* Nix/Guix)

Figure 5: Artifact longevity score for each of the three criteria (Table 2) - three top graphs, and the overall longevity score for the artifact - bottom graph. We recommend a minimum longevity score of 3 to award the Artifact Longevity Badge. Given the overall longevity scores of the 154 surveyed articles with available artifacts (Section 4.1), only 1.2 percent receive the Artifact Longevity Badge.

Threats to study's validity. This study focused only on five conferences on parallel and distributed systems, all from the *same* year (2023). Only one author of this study manually collected the data, which could have introduced errors and bias. The same author invested 5-10 minutes for each article to collect the aforementioned information, and may have missed details about the artifacts.

7 SUMMARY AND PERSPECTIVES

Summary. Awarding reproducibility badges to authors for their reproducibility efforts is an effective practice to encourage sharing of work and improving its quality. However, the notion of "reproducibility" considered by the badges is limited and does not cover important aspects of the reproducibility crisis. In this work, we surveyed 296 articles from five leading conferences in parallel and distributed systems (CCGrid, EuroSys, OSDI, PPoPP, SC) of 2023. For each article, we collected information on its artifact and the badges awarded. We conclude that the state of practice does not address the reproducibility problems in terms of *longevity* of the artifacts. Thus, we proposed a new badge to reward artifacts that will withstand the test of time. We associate with this new badge a framework for grading and awarding the badge when artifacts meet certain thresholds.

Perspectives. This work has the potential to trigger a series of longitudinal artifact reproducibility studies along different dimensions, such as workflow managers. Collecting data manually is slow and error-prone. Describing the metadata of the artifacts in a standardized format (similar to the Software Bill of Materials (SBOM) [79, 86]) would greatly improve the provenance of the artifacts. Such metadata would also allow automatic downloading and processing of artifacts, enabling artifact reproducibility studies at very large scales. Currently, each artifact longevity criterion has equal weight in the overall longevity score. A community discussion is needed to identify the adequate weight to attribute to each criterion.

ACKNOWLEDGMENTS

This research was funded, in whole or in part, by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 957407 as DAPHNE.

REFERENCES

[1] ACM. Artefact review badging. https://www.acm.org/publications/policies/artifact-review-badging. Accessed: 2023-04-04.

[2] P. Alliez, R. Di Cosmo, B. Guedj, A. Girault, M.-S. Hacid, A. Legrand, and N. Rougier. Attributing and referencing (research) software: best practices and outlook from inria. *Computing in Science & Engineering*, 22(1):39–52, 2019.

[3] P. Amstutz, M. R. Crusoe, N. Tijanić, B. Chapman, J. Chilton, M. Heuer, A. Kartashov, D. Leehr, H. Ménager, M. Nedeljkovich, et al. Common workflow language, v1. 0, 2016.

[4] M. Baker. 1,500 scientists lift the lid on reproducibility. *Nature*, 533(7604):452–454, May 2016. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/533452a. URL: http://www.nature.com/doifinder/10.1038/533452a (visited on 05/03/2019).

[5] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, editors, *Cloud Computing and Services Science*. Volume 367, Communications in Computer and Information Science, pages 3–20. Springer International Publishing, 2013. ISBN: 978-3-319-04518-4. DOI: 10.1007/978-3-319-04519-1_1.

[6] L. A. Barba. Terminologies for reproducible research. *arXiv preprint arXiv:1802.03311*, 2018.

[7] M. Beller. Why i will never join an artifacts evaluation committee again. *Inventitech. com. https://inventitech. com/blog/why-i-will-never-review-artifacts-again/(Accessed: Feb. 9, 2022)*, 2020.

[8] BIPM, IEC, IFCC, ILAC, IUPAC, IUPAP, ISO, OIML. International vocabulary of metrology – basic and general concepts and associated terms (vim) 3rd edition. URL: %5Curl%7Bhttps://www.bipm.org/documents/20126/2071204/JCGM_200_2012.pdf/f0e1ad45-d337-bbeb-53a6-15fe649d0ff1%7D.

[9] blinry. Building 15-year-old software with nix. https://blinry.org/nix-time-travel/. Accessed: 2023-04-16.

[10] G. R. Brammer, R. W. Crosby, S. J. Matthews, and T. L. Williams. Paper mâché: creating dynamic reproducible science. *Procedia Computer Science*, 4:658–667, 2011.

[11] A. Brinckman, K. Chard, N. Gaffney, M. Hategan, M. B. Jones, K. Kowalik, S. Kulasekaran, B. Ludäscher, B. D. Mecum, J. Nabrzyski, et al. Computing environments for reproducibility: capturing the "whole tale". *Future Generation Computer Systems*, 94:854–867, 2019.

[12] W. z. Castell, D. Dransch, G. Juckeland, M. Meistring, B. Fritzsch, R. Gey, B. Höpfner, M. Köhler, C. Meeßen, H. Mehrtens, et al. Towards a quality indicator for research data publications and research software publications–a vision from the helmholtz association. *arXiv preprint arXiv:2401.08804*, 2024.

[13] [Software] cea-hpc, 2024. URL: https://github.com/cea-hpc/modules, SWHID: ⟨swh:1:dir:a83ee4b9f7d1a2f6377326dd16ac839450a6e6fc;origin=https://github.com/cea-hpc/modules;⟩.

[14] C. Cloud. The cc-snapshot utility. https://chameleoncloud.readthedocs.io/en/latest/technical/images.html#the-cc-snapshot-utility. Accessed: 2023-04-03.

[15] C. Collberg, T. Proebsting, and A. M. Warren. Repeatability and Benefaction in Computer Systems Research - A Study and a Modest Proposal. en:68, 2015.

[16] M. Colom, B. Kerautret, N. Limare, P. Monasse, and J.-M. Morel. Ipol: a new journal for fully reproducible research; analysis of four years development. In *2015 7th International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5. IEEE, 2015.

[17] CORE. International core conference rankings: icore. URL: https://www.core.edu.au/icore-portal.

[18] L. Courtès. Connecting reproducible deployment to a long-term source code archive, march 2019. URL: https://guix.gnu.org/en/blog/2019/connecting-reproducible-deployment-to-a-long-term-source-code-archive/.

[19] L. Courtès. Functional Package Management with Guix. en. *arXiv:1305.4584 [cs]*, May 2013. URL: http://arxiv.org/abs/1305.4584 (visited on 06/13/2020).

[20] R. F. da Silva, R. M. Badia, V. Bala, D. Bard, P.-T. Bremer, I. Buckley, S. Caino-Lores, K. Chard, C. Goble, S. Jha, D. S. Katz, D. Laney, M. Parashar, F. Suter, N. Tyler, T. Uram, I. Altintas, S. Andersson, W. Arndt, J. Aznar, J. Bader, B. Balis, C. Blanton, K. R. Braghetto, A. Brodutch, P. Brunk, H. Casanova, A. C. Lierta, J. Chigu, T. Coleman, N. Collier, I. Colonnelli, F. Coppens, M. Crusoe, W. Cunningham, B. de Paula Kinoshita, P. D. Tommaso, C. Doutriaux, M. Downton, W. Elwasif, B. Enders, C. Erdmann, T. Fahringer, L. Figueiredo, R. Filgueira, M. Foltin, A. Fouilloux, L. Gadelha, A. Gallo, A. G. Saez, D. Garijo, R. Gerlach, R. Grant, S. Grayson, P. Grubel, J. Gustafsson, V. Hayot-Sasson, O. Hernandez, M. Hilbrich, A. Justine, I. Laflotte, F. Lehmann, A. Luckow, J. Luettgau, K. Maheshwari, M. Matsuda, D. Medic, P. Mendygral, M. Michalewicz, J. Nonaka, M. Pawlik, L. Pottier, L. Pouchard, M. Putz, S. K. Radha, L. Ramakrishnan, S. Ristov, P. Romano, D. Rosendo, M. Ruefenacht, K. Rycerz, N. Saurabh, V. Savchenko, M. Schulz, C. Simpson, R. Sirvent, T. Skluzacek, S. Soiland-Reyes, R. Souza, S. R. Sukumar, Z. Sun, A. Sussman, D. Thain, M. Titov, B. Tovar, A. Tripathy, M. Turilli, B. Tuznik, H. van Dam, A. Vivas, L. Ward, P. Widener, S. Wilkinson, J. Zawalska, and M. Zulfiqar. Workflows Community Summit 2022: A Roadmap Revolution, Mar. 2023. DOI: 10.5281/zenodo.7750670. URL: https://doi.org/10.5281/zenodo.7750670.

[21] Debian. Snapshot. http://snapshot.debian.org/. Accessed: 2024-02-09.

[22] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. F. Da Silva, M. Livny, et al. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015.

[23] H. P. de León. Creating successful artifacts. https://hernanponcedeleon.github.io/articles/artifacts.html. Accessed: 2024-02-08.

[24] R. Di Cosmo and S. Zacchiroli. Software Heritage: Why and How to Preserve Software Source Code. In *iPRES 2017 - 14th International Conference on Digital Preservation*, pages 1–10, Kyoto, Japan, Sept. 2017. URL: https://hal.archives-ouvertes.fr/hal-01590958.

[25] R. Di Cosmo and S. Zacchiroli. Software heritage: why and how to preserve software source code. In *iPRES 2017-14th International Conference on Digital Preservation*, pages 1–10, 2017.

[26] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame. Nextflow enables reproducible computational workflows. *Nature biotechnology*, 35(4):316–319, 2017.

[27] Docker. Optimizing builds with cache management. https://docs.docker.com/build/cache/. Accessed: 2023-06-20.

[28] E. Dolstra, M. de Jonge, and E. Visser. Nix: A Safe and Policy-Free System for Software Deployment. en:14, 2004.

[29] E. Dolstra and A. Löh. Nixos: a purely functional linux distribution. *SIGPLAN Not.*, 43(9):367–378, Sept. 2008. ISSN: 0362-1340. DOI: 10.1145/1411203.1411255. URL: https://doi.org/10.1145/1411203.1411255.

[30] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019. URL: https://www.flux.utah.edu/paper/duplyakin-atc19.

[31] D. G. Feitelson. From Repeatability to Reproducibility and Corroboration. en. *ACM SIGOPS Operating Systems Review*, 49(1):3–11, Jan. 2015. ISSN: 0163-5980. DOI: 10.1145/2723872.2723875. URL: https://dl.acm.org/doi/10.1145/2723872.2723875 (visited on 05/21/2020).

[32] figshare. Figshare. https://figshare.com/. Accessed: 2024-01-23.

[33] E. Frachtenberg. Research artifacts and citations in computer systems papers. *PeerJ Computer Science*, 8:e887, 2022.

[34] J. L. Furlani. Modules: providing a flexible user environment. In *Proceedings of the fifth large installation systems administration conference (LISA V)*, pages 141–152, 1991.

[35] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. The Spack package manager: bringing order to HPC software chaos. en. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Austin Texas. ACM, Nov. 2015. ISBN: 978-1-4503-3723-6. DOI: 10.1145/2807591.2807623. URL: https://dl.acm.org/doi/10.1145/2807591.2807623 (visited on 11/22/2021).

[36] Github. Issuing a persistent identifier for your repository with zenodo. https://docs.github.com/en/repositories/archiving-a-github-repository/referencing-and-citing-content. Accessed: 2024-01-24.

[37] Globus. Globus website. https://globus.org/. Accessed: 2024-01-28.

[38] Google. Bidding farewell to google code. https://opensource.googleblog.com/2015/03/farewell-to-google-code.html. Accessed: 2024-02-05.

[39] [Software] Gricad, Nix Ciment Channel 2023. URL: https://github.com/Gricad/nix-ciment-channel, SWHID: ⟨swh:1:dir:dec8c22b23ba51650f65352fa2fc2640f1532bea;origin=https://github.com/Gricad/nix-ciment-channel⟩.

[40] Grid'5000. Creating an environment images using tgz-g5k. https://grid5000.fr/w/Environment_creation#Creating_an_environment_images_using_tgz-g5k. Accessed: 2023-04-03.

[41] Q. Guilloteau, J. Bleuzen, M. Poquet, and O. Richard. Painless transposition of reproducible distributed environments with nixos compose. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12. IEEE, 2022.

[42] Q. Guilloteau, F. M. Ciorba, M. Poquet, D. Goepp, and O. Richard. Dataset for the Paper "Longevity of Artifacts in Leading Parallel and Distributed Systems Conferences: a Review of the State of the Practice in 2023". Zenodo, Feb. 2024. DOI: 10.5281/zenodo.10650804. URL: https://doi.org/10.5281/zenodo.10650804.

[43] [Software] GuilloteauQ, 2024. URL: https://github.com/GuilloteauQ/artefact-lifetime, SWHID: ⟨swh:1:dir:3904795a49327b1e9b7ef1a8aff995b1f94ae6fb;origin=https://github.com/GuilloteauQ/artefact-lifetime;visit=swh:1:snp:9adc667f9fefcb6783df076520ef357231030ad4;anchor=swh:1:rev:ad4d19abb5da12fd6df708bc13e444d834c85dfa⟩.

[44] [Software] Guix-HPC, Guix-HPC 2023. URL: https://gitlab.inria.fr/guix-hpc/guix-hpc, SWHID: ⟨swh:1:dir:aabd69b989444999630729faf0184f4d68ff13fa;origin=https://gitlab.inria.fr/guix-hpc/guix-hpc⟩.

[45] S. Heritage. Software heritage. https://www.softwareheritage.org/. Accessed: 2023-03-30.

[46] B. Hermann. What has artifact evaluation ever done for us? *IEEE Security & Privacy*, 20(5):96–99, 2022.

[47] 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*, pages 469–480, 2020.

[48] R. Heumüller, S. Nielebock, J. Krüger, and F. Ortmeier. Publish or perish, but do not forget your software artifacts. *Empirical Software Engineering*, 25(6):4585–4616, 2020.

[49] https://artifact-eval.org. Guidelines for packaging aec submissions. https://artifact-eval.org/guidelines.html. Accessed: 2024-02-08.

[50] S. Hunold. A survey on reproducibility in parallel computing. *arXiv preprint arXiv:1511.04217*, 2015.

[51] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[52] M. C. Kidwell, L. B. Lazarević, E. Baranski, T. E. Hardwicke, S. Piechowski, L.-S. Falkenberg, C. Kennett, A. Slowik, C. Sonnleitner, C. Hess-Holden, et al. Badges to acknowledge open practices: a simple, low-cost, effective method for increasing transparency. *PLoS biology*, 14(5):e1002456, 2016.

[53] J. Köster and S. Rahmann. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 2012.

[54] D. Méndez Fernández, W. Böhm, A. Vogelsang, J. Mund, M. Broy, M. Kuhrmann, and T. Weyer. Artefacts in software

engineering: a fundamental positioning. *Software & Systems Modeling*, 18:2777–2786, 2019.

[55] M. Mercier, A. Faure, and O. Richard. Considering the development workflow to achieve reproducibility with variation. In *SC 2018-Workshop: ResCuE-HPC*, pages 1–5, 2018.

[56] T. Moreau, M. Massias, A. Gramfort, P. Ablin, P.-A. Bannier, B. Charlier, M. Dagréou, T. Dupre la Tour, G. Durif, C. F. Dantas, et al. Benchopt: reproducible, efficient and collaborative optimization benchmarks. *Advances in Neural Information Processing Systems*, 35:25404–25421, 2022.

[57] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices*, 44(3):265–276, 2009.

[58] I. Newton. Letter from sir isaac newton to robert hooke.

[59] NixOS. Nixpkgs. https://github.com/nixos/nixpkgs. Accessed: 2023-04-04.

[60] L. Nussbaum. Testbeds support for reproducible research. In *Proceedings of the reproducibility workshop*, pages 24–26, 2017.

[61] Nvidia. Container support policy. https://archive.softwareheritage.org/swh:1:cnt:167b243e20f4f7e38efcd1c9d1696f291de6c5e0;origin=https://gitlab.com/nvidia/container-images/cuda;visit=swh:1:snp:62fbbe6c6441981445c19b0632f4bc0c69736d12;anchor=swh:1:rev:e3ff10eab3a1424fe394899df0e0f8ca5a410f0f;path=/doc/support-policy.md. Accessed: 2024-01-24.

[62] [Software] OAR-Team, NUR-Kapack 2023. URL: https://github.com/oar-team/nur-kapack, SWHID: ⟨swh:1:dir:1a6970dbce78a86062648a7c76978f674f136607;origin=https://github.com/oar-team/nur-kapack⟩.

[63] R. Padhye. Artifact evaluation: tips for authors. https://blog.padhye.org/Artifact-Evaluation-Tips-for-Authors/. Accessed: 2024-02-08.

[64] C. Pérignon, K. Gadouche, C. Hurlin, R. Silberman, and E. Debonnel. Certify reproducibility with confidential data. *Science*, 365(6449):127–128, 2019.

[65] H. E. Plesser. Reproducibility vs. replicability: a brief history of a confused terminology. *Frontiers in neuroinformatics*, 11:76, 2018.

[66] D. Randall and C. Welser. *The Irreproducibility Crisis of Modern Science. Causes, Consequences, and the Road to Reform.* en. National Association of Scholars, New York, 2018. URL: https://www.nas.org/reports/the-irreproducibility-crisis-of-modern-science.

[67] Repology. Repology. https://repology.org/. Accessed: 2024-02-09.

[68] D. Rosendo, K. Keahey, A. Costan, M. Simonin, P. Valduriez, and G. Antoniu. Kheops: cost-effective repeatability, reproducibility, and replicability of edge-to-cloud experiments. In *Proceedings of the 2023 ACM Conference on Reproducibility and Replicability*, pages 62–73, 2023.

[69] D. Rosendo, P. Silva, M. Simonin, A. Costan, and G. Antoniu. E2Clab: Exploring the Computing Continuum through Repeatable, Replicable and Reproducible Edge-to-Cloud Experiments. In *Cluster 2020 - IEEE International Conference on Cluster Computing*, pages 1–11, Kobe, Japan, Sept. 2020. DOI: 10.1109/CLUSTER49012.2020.00028. URL: https://hal.science/hal-02916032.

[70] N. P. Rougier and K. Hinsen. Rescience c: a journal for reproducible replications in computational science. In *Reproducible Research in Pattern Recognition: Second International Workshop, RRPR 2018, Beijing, China, August 20, 2018, Revised Selected Papers 2*, pages 150–156. Springer, 2019.

[71] A. Rowhani-Farid, M. Allen, and A. G. Barnett. What incentives increase data sharing in health and medical research? a systematic review. *Research integrity and peer review*, 2(1):1–10, 2017.

[72] C. Ruiz, S. Harrache, M. Mercier, and O. Richard. Reconstructable Software Appliances with Kameleon. en. *ACM SIGOPS Operating Systems Review*, 49(1):80–89, Jan. 2015. ISSN: 0163-5980. DOI: 10.1145/2723872.2723883. URL: https://dl.acm.org/doi/10.1145/2723872.2723883 (visited on 06/12/2020).

[73] S. Sharma, A. Hussain, and H. Saran. Towards repeatability & verifiability in networking experiments: a stochastic framework. *Journal of Network and Computer Applications*, 81:12–23, 2017.

[74] A. Silver. Software simplified. *Nature*, 546(7656):173–174, 2017.

[75] A. M. Smith, K. E. Niemeyer, D. S. Katz, L. A. Barba, G. Githinji, M. Gymrek, K. D. Huff, C. R. Madan, A. C. Mayes, K. M. Moerman, et al. Journal of open source software (joss): design and first-year review. *PeerJ Computer Science*, 4:e147, 2018.

[76] V. Stodden and M. S. Krafczyk. Assessing reproducibility: an astrophysical example of computational uncertainty in the hpc context. In *Proceedings of the 1st Workshop on Reproducible, Customizable and Portable Workflows for HPC at SC*, volume 18, 2018.

[77] F. Strozzi, R. Janssen, R. Wurmus, M. R. Crusoe, G. Githinji, P. Di Tommaso, D. Belhachemi, S. Möller, G. Smant, J. de Ligt, et al. Scalable workflows and reproducible data analysis for genomics. *Evolutionary Genomics: Statistical and Computational Methods*:723–745, 2019.

[78] [Software] tdurieux, 2024. URL: https://github.com/tdurieux/anonymous_github, SWHID: ⟨swh:1:dir:9239038ce7c56d17b6b3ee3acd6d5ef86da038c1;origin=https://github.com/tdurieux/anonymous_github⟩.

[79] N. Telecommunications and I. Administration. Software bill of materials. https://www.ntia.gov/page/software-bill-materials. Accessed: 2024-02-05.

[80] TPDS. Reproducibility initiative. URL: https://www.computer.org/csdl/journal/td/write-for-us/104303.

[81] Tweag.io. What problems do flakes solve? https://www.tweag.io/blog/2020-05-25-flakes/. Accessed: 2023-04-04.

[82] UNESCO. Understanding open science. DOI: 10.54677/UTCD9302.

[83] J. Vivian, A. A. Rao, F. A. Nothaft, C. Ketchum, J. Armstrong, A. Novak, J. Pfeil, J. Narkizian, A. D. Deran, A. Musselman-Brown, et al. Toil enables reproducible, open source, big biomedical data analyses. *Nature biotechnology*, 35(4):314–316, 2017.

[84] 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*, pages 145–156, 2022.

[85] L. Wratten, A. Wilm, and J. Göke. Reproducible, scalable, and shareable analysis pipelines with bioinformatics workflow managers. *Nature methods*, 18(10):1161–1168, 2021.

[86] B. Xia, T. Bi, Z. Xing, Q. Lu, and L. Zhu. An empirical study on software bill of materials: where we stand and the road ahead. *arXiv preprint arXiv:2301.05362*, 2023.

[87] zenodo. Zenodo. https://zenodo.org/. Accessed: 2023-03-30.

A ARTIFACT DESCRIPTION

We recommend that the reader refers to the `README` of the repository.

A.1 Repository

The snapshot of the repository containing the sources for the analysis scripts, the forms for each of the papers surveyed, and the sources for the paper is available on Software-Heritage [43]:

URL: https://archive.softwareheritage.org/swh:1:dir:3904795a49327b1e9b7ef1a8aff995b1f94ae6fb;origin=https://github.com/GuilloteauQ/artefact-lifetime;visit=swh:1:snp:9adc667f9fefcb6783df076520ef357231030ad4;anchor=swh:1:rev:ad4d19abb5da12fd6df708bc13e444d834c85dfa

Software-Heritage metadata:

```
swh:1:dir:3904795a49327b1e9b7ef1a8aff995b1f94ae6fb;
origin=https://github.com/GuilloteauQ/artefact-lifetime;
visit=swh:1:snp:9adc667f9fefcb6783df076520ef357231030ad4;
anchor=swh:1:rev:ad4d19abb5da12fd6df708bc13e444d834c85dfa
```

A.2 Dataset

The data set used in this article is available on Zenodo under DOI **10.5281/zenodo.10650804** [42].

A.3 Software dependencies

This artifact uses Nix to set up the software environment. We make use of the Flake feature of Nix, which was introduced in Nix 2.4.

A.4 Hardware dependencies

The only hardware requirement is a machine with an Internet connection and that is able to download Nix (Linux, MacOS). The reader might need to have root privileges on the machine for the installation process.

A.5 Estimated time to reproduce

Most of the time will be spent installing the dependencies. We estimate that the entire workflow will be completed in less than 30 minutes.

Received 12 February 2024; revised 12 March 2009; accepted 5 June 2009