

Reproducibility of Build Environments through Space and Time

Julien Malka
julien.malka@ens.fr
LTCI, Télécom Paris,
Institut Polytechnique de Paris
Palaiseau, France

Stefano Zacchiroli
stefano.zacchiroli@telecom-paris.fr
LTCI, Télécom Paris,
Institut Polytechnique de Paris
Palaiseau, France

Théo Zimmermann
theo.zimmermann@telecom-paris.fr
LTCI, Télécom Paris,
Institut Polytechnique de Paris
Palaiseau, France

ABSTRACT

Modern software engineering builds up on the composability of software components, that rely on more and more direct and transitive dependencies to build their functionalities. This principle of reusability however makes it harder to reproduce projects' build environments, even though reproducibility of build environments is essential for collaboration, maintenance and component lifetime. In this work, we argue that functional package managers provide the tooling to make build environments reproducible in space and time, and we produce a preliminary evaluation to justify this claim. Using historical data, we show that we are able to reproduce build environments of about 7 million Nix packages, and to rebuild 99.94% of the 14 thousand packages from a 6-year-old Nixpkgs revision.

ACM Reference Format:

Julien Malka, Stefano Zacchiroli, and Théo Zimmermann. 2024. Reproducibility of Build Environments through Space and Time. In *New Ideas and Emerging Results (ICSE-NIER'24)*, April 14–20, 2024, Lisbon, Portugal. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.1145/3639476.3639767>

1 INTRODUCTION

Modern software engineering is built on the principles of **composability** and **reusability**: instead of re-implementing existing functionalities, software typically depends on libraries providing them. This approach increases productivity and helps build more robust software: each component focuses on a little piece of functionality, and tries to do it well. However, as a consequence, software projects accumulate (direct and transitive) dependencies, resulting in an increasingly complex software supply chain [15]. **Build environments**—that is, all the components and variables, up to their specific versions, necessary to build the software—become more complex and difficult to reproduce in space and time, a situation often referred to as *dependency hell* [9].

We say that a build environment is reproducible *in space* when it is possible to obtain an environment containing the exact same components and variables on another machine. We say that it is reproducible *in time*, when this property does not decay over time, which means one can obtain a build environment that remains identical to the one that was used when building a given software version in the past. Reproducibility of the build environment is a desirable property for a software project: in space, it allows for smoother collaboration between developers that can then work on

the project in identical conditions, thus alleviating barriers to contribution. It also facilitates bug reproduction by projects maintainers. Reproducibility of the build environment in time allows rebuilding past software versions, which is very useful to better understand how software used to work or to bisect bugs [4, 7]. It also helps combat the risk of software dying because progressive independent evolution of its dependencies makes it practically impossible to find a combination of dependency versions that allows rebuilding it.

A number of package managers have taken steps towards improving the reproducibility of their build environments by introducing *lockfiles* that contain a reference to the exact versions of dependencies used by the project (e.g., npm with `package-lock.json` [13]). Although they do help, these methods are insufficient for projects that have dependencies beyond a single ecosystem. Another popular approach is to provide a container image, allowing users to reproduce the same build environment by downloading the image and running the container. However, making the process of generating these images reproducible itself requires care [25].

We believe that making build environments easily reproducible could have decisive impact on the practice of software engineering by facilitating collaborative development, helping maintenance and reinforcing software composability by combatting dependency aversion. We also aim to demonstrate that functional package managers, like Nix [10] or Guix [5], are the right tools for this task. While the Nix and Guix communities claim that they provide the necessary tooling to achieve reproducibility of build environments [28], no empirical evidence has been available thus far to support these claims. In this work, we empirically evaluate whether space and time reproducibility of build environments is achievable using the Nix package manager (**RQ1**) and if that allows rebuilding past software versions (**RQ2**). While making build environments reproducible may help perform *bit-by-bit* reproducible builds [16], the study of Nix abilities for that purpose is left for future work.

Our results show that we can achieve 99.99% reproducibility of build environments over 7 010 516 packages coming from 200 historical revisions of Nixpkgs, the Nix package set. Additionally, we were able to rebuild 99.94% of the packages from a 6-year-old Nixpkgs revision, demonstrating that reproducibility of build environments is actually useful for software rebuildability.

2 CONTEXT & DEFINITIONS

Functional Package Manager (FPM). Nix and Guix are implementations of a package deployment model, first introduced by Dolstra [10], that is conceptually very different from most other package managers. In FPMs, packages are distributed as **pure functions** of their build- and run-time dependencies. In Nix, for example, packages are specified as expressions in the Nix language. Figure 1 shows

```

{ stdenv, fetchurl, ncurses }: [ 1 ]
stdenv.mkDerivation rec { [ 2 ]
  pname = "nano";
  version = "7.2";
  src = fetchurl { [ 3 ]
    url = "mirror://gnu/nano/${pname}-${version}.tar.xz";
    sha256 = "hvNEJ2i9KHPOxpP4PN+AtLRErTzBR2C3Q2FHT8h6RSY=";
  };
  buildInputs = [ ncurses ]; [ 4 ]
  configureFlags = [ "--sysconfdir=/etc" ]; [ 5 ]
}

```

Figure 1: Nix expression of nano, modified for readability.

```

{
  "args": [ "-e", "/nix/store/6xg25947...-default-builder.sh" ], [ 6 ]
  "builder": "/nix/store/rhvbjmcfc...-bash-5.2-p15/bin/bash",
  "env": {
    "buildInputs": "/nix/store/9jmgys8b...-ncurses-6.4-dev",
    "cmakeFlags": "",
    "configureFlags": "--sysconfdir=/etc",
    "name": "nano-7.2"
  },
  "inputDrvs": { [ 7 ]
    "/nix/store/3qsdhv4v...-stdenv-linux.drv": [ "out" ],
    "/nix/store/asq3sjwr...-nano-7.2.tar.xz.drv": [ "out" ],
    "/nix/store/had1mg70...-bash-5.2-p15.drv": [ "out" ],
    "/nix/store/q56mxcpc...-ncurses-6.4.drv": [ "dev" ]
  },
  "inputSrcs": [ "/nix/store/6xg25947...-default-builder.sh" ],
  "outputs": {
    "out": { "path": "/nix/store/385vk5j4...-nano-7.2" } [ 8 ]
  },
}

```

Figure 2: Derivation of nano, modified for readability.

a Nix expression for the nano text editor. This is a function whose inputs [1] are: `stdenv` (minimal build environment), `fetchurl` (function to download the program sources), and `ncurses` (a build-time dependency). The output of the `stdenv.mkDerivation` function [2] is a **derivation**, the intermediate representation Nix will use to build the package. The derivation is constructed by passing several arguments to the `mkDerivation` function: the `src` parameter [3] contains the source of the software up to its specific version (Nix checks that the correct version has been downloaded by comparing the hash of the content with the specified hash). The build-time dependencies are specified in the `buildInputs` list [4]. The `ncurses` object is also a Nix derivation, that Nix will build *before* nano. A Nix derivation is simply a *build recipe*, that Nix will use to create a *build environment* with the source and `buildInputs` available in it. It will then run a bash script called the *builder* to produce the final artifact. It is possible to provide a custom builder, but here the default autoconf builder is used, and some flags are passed in [5].

The process by which Nix creates a derivation from a Nix expression is called **evaluation**. The resulting derivation can then be used to build a program. The outputs of both the evaluation phase and the build phase are stored in the special `/nix/store` directory of the build host. The name of derivation files contain a cryptographic hash that is computed based on its contents (including the precise version of the program sources, but also of all its dependencies).

This use of cryptographic hashes for paths in the Nix store allows it to contain many versions of the same package.

Figure 2 shows an extract of the content of the derivation obtained by evaluating the previous Nix expression. It contains all the information derived from the Nix file. We can find the path to the builder [6], the list of the derivations on which nano depends [7], including one for the program sources, and the `/nix/store` path where the build process will create its outputs [8], which also contains a cryptographic hash that depends on the exact dependency versions and build parameters. For details about how the outputs hashes are computed see Dolstra [10].

Derivations can be used in two ways: they can be instantiated, which means running the builder in the specified (hermetic and sandboxed) build environment to produce a build output in the Nix store, or they can be used to spawn a build environment where developers can then manually run their builds. One of the main claims associated with FPMs is that this build environment will be highly reproducible and that it will therefore allow performing the same build reliably (from one machine to another, and over time).

Nixpkgs, the Nix package collection. The users of Nix have collaboratively created a collection of Nix expressions to build various pieces of software. This collection forms the basis of the NixOS Linux distribution [11], but it is available beyond NixOS, to any Nix user. With over 80 000 packages, Nixpkgs is, at the time of writing, the largest Linux distribution in number of packages [21], as it repackages many pieces of software coming from application-specific (e.g., Emacs, VS Code) and programming-language specific (e.g., Haskell, Python) ecosystems. Nixpkgs provides several “channels”, including a rolling-release called `nixpkgs-unstable`.

Store substitution mechanism. While Nixpkgs is a source-based software distribution, Nix allows to substitute build outputs resulting from the build process specified in derivations with pre-built outputs provided by a binary cache. A binary cache is a large dictionary linking output paths to compressed build outputs. When Nix is configured to use a binary cache (a.k.a., a substituter), Nix will pause at the end of the evaluation phase and query the binary cache for the output paths of the obtained derivations. When such output paths are available in the binary cache, Nix will then download and unpack them in the Nix store, instead of running the derivation build process. It will then proceed to build the remaining derivations, for which no cached artifact was available.

Hydra, the Nix continuous integration platform. The Nixpkgs distribution comes with an official binary cache, `cache.nixos.org`, which is populated by Hydra, a continuous integration platform. At regular intervals of time, Hydra evaluates the current revision of the Nixpkgs git repository. This evaluation results in a list of derivations (otherwise called *jobs*) to build. Hydra then builds all of these jobs, unless they produce an output path already in cache. If the build is successful for a pre-defined list of important jobs, Hydra then pushes the build outputs to the official binary cache and updates the `nixpkgs-unstable` channel.

3 METHODOLOGY

In this section we describe the methodology followed to answer the stated research questions.

3.1 RQ1: Reproducibility of build environments

To evaluate whether Nix build environments are reproducible in space and time, we focus on the Nix evaluation step. As explained before, evaluation is the process by which Nix transforms an expression written by humans into a machine-readable derivation, that defines exactly how to create a build environment, the versions of dependencies to put in scope, and how to set environment variables. Nix users can then use the derivation to spawn a build environment, or let Nix perform the build of a known software version.

Using historical data coming from Hydra, we are interested in assessing whether we can reproduce: the exact same list of jobs resulting from the evaluation of a given Nixpkgs revision (**RQ1.1**); and identical derivations for each of these jobs (**RQ1.2**).

Comparing derivations that we obtain locally with the historical ones from Hydra is actually difficult, because Hydra does not make available the derivations it built (it only pushes build outputs to the binary cache). Sometimes, we can get the path of the derivation, and that would be sufficient to check if the locally built and Hydra derivations are identical or differ, since the path contains a cryptographic hash computed from the content of the derivation. But even this path is not always available, because Hydra does not keep its trace when it did not rebuild a job (whose output was already in cache). Besides, it can happen that two derivations with a different hash share the same output path. Indeed, derivation hashes incorporate more information on the build process than output paths (including, e.g., what `curl` version to use to download the program sources). This is because the computation of output paths is designed to create identical paths when differing derivations are guaranteed to produce the same result. Given the way output paths are computed, identical output paths should still ensure identical build environments. Therefore, we adjust **RQ1.2** slightly into: assessing whether we can reproduce identical output paths for all the jobs (as we can always retrieve historical Hydra output paths).

We perform our experiment on a sample of the Nixpkgs revisions. We start from a dataset (available at channels.nix.gsc.io), which contains more than 2200 revisions belonging to the `nixpkgs-unstable` channel spanning from 2017 to 2023, with in average 23 hours of separation between each of them. To keep our study computationally reasonable, we extract 200 revisions from this dataset, keeping at the same time the maximum time spread possible and a regular spacing between the selected revisions. The result of the sampling operation is a set of 200 Nixpkgs revisions that:

- have been promoted to the `nixpkgs-unstable` channel;
- span from 2017 to 2023;
- have a regular spacing of 10 days and 20 hours on average.

We evaluate the selected revisions using a variant of the Nix evaluator called `nix-eval-jobs` [14], a piece of software derived from Hydra's component in charge of the evaluation phase. It allows for faster evaluation of complete Nixpkgs revisions than the built-in Nix evaluator as it can evaluate several Nix derivations in parallel, and, contrary to the built-in Nix evaluator, it will not fail because some jobs fail to evaluate. This is important when building entire Nixpkgs revisions as there are always a few jobs that fail to evaluate, sometimes on purpose (e.g., because they have security issues).

We scrape Hydra's website to obtain historical results to compare to. For each sampled revision, we scrape from the associated Hydra

webpage the list of jobs that succeeded to evaluate, and for each of them, we scrape the job webpage to retrieve its output path(s).

3.2 RQ2: Rebuilding past software versions

We are interested in understanding if achieving build environment reproducibility is sufficient to confidently rebuild past software versions. To answer this, we build all the jobs from our most ancient Nixpkgs revision (from 2017, which is the oldest `nixpkgs-unstable` revision available in our dataset). It contains 14 753 jobs for the `x86_64-linux` architecture, out of them 14 461 built successfully at the time. We compare the build status stored in Hydra to our local build success or failure. In case a build fails in our experiment while it had succeeded on Hydra in 2017, we look at the build log to try to understand the cause of the failure.

Note that, as this step is much more computationally intensive than the previous one (we are actually building an entire distribution of open source software), our sample size is much smaller. Instead of evaluating the reproducibility over time on a large sample of Nixpkgs revision, we focus on the oldest revision as we expect that, in principle, the farther away we go in the past, the more difficult it becomes to rebuild software. Therefore, we expect that if we obtain good results on this old revision, they should reasonably extend to more recent revisions. We recognize that there are threats to this claim of external validity, e.g., if changes in software practice or in the scope of Nixpkgs trigger an increase in flaky builds.

Besides, for now, we do not check for *bit-by-bit* reproducible builds [16]. Extending our evaluation to build more revisions and checking for reproducible builds are both part of our future plans.

We use the Nix cache extensively in this step. This allows building many jobs in parallel, by focusing on rebuilding the selected job and not its dependencies. While this helps answer the question "Is build environment reproducibility sufficient for rebuilding past software versions?", it does not answer the question "Can we rebuild an entire past revision of Nixpkgs without relying on cache?". The latter would be a much more difficult achievement because failures would induce cascade effects on all their dependents, but also because of the risk of program sources becoming unavailable. Currently, these sources are still provided through the Nix cache.

4 RESULTS

In this section, we describe the results obtained by performing our experiments.

4.1 RQ1: Reproducibility of build environments

During the first phase of our experiments to assess the reproducibility of the evaluation of Nixpkgs revisions, we discovered two bugs that resulted in obtaining different lists of jobs. One was a bug of `nix-eval-jobs`, which did not follow the current convention in Nixpkgs and Hydra to determine when to include a job or not in the result of the evaluation. We fixed this bug (see [18]) and we used the updated version to perform our experiments. The other was a bug of Hydra, which skipped jobs which contained a dot in their name (which is quite rare in Nixpkgs). We also fixed this bug (see [19]), but we had to take this bug into account when performing our comparisons with historical Hydra data.

Up to the discrepancy coming from this second bug, our results show that we can obtain 100% identical lists of jobs from our sample of historic Nixpkgs revisions. When comparing jobs' output paths, we obtain 99.99% identical output paths. For each revision, there were at most 4 expressions causing jobs with differing output paths and they all used Nix features that cause impure evaluation (`builtins.nixVersion` or `lib.inNixShell` for example). These results show that Nix is able to achieve perfect build environment reproducibility, given an unchanged Nix expression (avoiding impure features), and despite several differences in the conditions of the evaluation: different hardware, Linux distribution (NixOS for the Hydra infrastructure, Ubuntu for our experiments), different Nix versions (we used Nix version 2.6 in our experiments, while the Nix version used in Hydra has evolved over the years), and points in time (between 0- and 6-year differences).

4.2 RQ2: Rebuilding past software versions

We were able to successfully rebuild 14 452 out of the 14 461 jobs that Hydra had successfully built from our selected revision (5328102), giving us a success rate of 99.94%. We performed a preliminary assessment of the reasons for the failures of the 9 remaining jobs, and confirming or adjusting this assessment is part of our future plans. At this point, we have classified the jobs into 3 classes of reasons that might cause the failure.

Current build sandbox leakage. 3 jobs failed because the build script rejected the kernel version or the OS used, pieces of information that should not be available inside the sandbox.

Flaky tests. 1 job failed because of a single failed test, which makes us suspect a flaky test (a test that fails inconsistently) [17, 26].

Past build sandbox leakage. For several other jobs, including 1 whose tests fail because of an expired certificate, and 2 whose failures look related to the shell, we suspect that the failure could come from a stricter build sandbox in newer Nix versions, preventing the build to have access to unspecified dependency or data that would have previously been available from the environment.

5 RELATED WORK

“Moving parts” in build environments have been recognized as problematic for a long time, because they lead to non-reproducibility issues. Continuous Integration (CI) [12, 23], a key DevOps practice, expects build environment stability. CI build failures have been studied empirically and at a large scale [29, 32, 37]. It is well-established that, independently of the programming language, the most common cause of CI build failures are (bloated) dependency issues [34]. More generally, the so-called “dependency hell” [9] is a major factor in the non-reproducibility of development environments. For example, Mukherjee et al. [24] investigated how this is the case in the Python ecosystem; Zampetti et al. [36] confirm this in a broader study of “CI smells”. Abate et al. [1, 13] show how failed dependency resolution is a recurrent cause of package non-installability across different package ecosystems. Flaky tests [17, 26] are equally annoying for developers and can also be caused by the unexpected displacements of build environment parts, including dependencies. Dependency pinning, as supported by package manager “lockfiles”,

is a partial solution to the problem [13], which does not address reproducibility causes other than moving dependencies.

In 2020, the ReScienceC journal ran the Ten Years Reproducibility Challenge [27, 30], defying scientists into rerunning software associated to their own papers published at least 10 years prior. Participants generally succeeded, but reported about significant difficulties in doing so. Supporting reproducible science via software tooling is currently a hot topic, with researchers looking into how to leverage Docker for that [2, 3], but also functional package managers [6, 35]. One of the promises of functional package management [5, 10] is indeed to fully describe build environments, removing dependency issues from the equation of build and test failures. This is the *theory*, at least, but one that to the best of our knowledge had never been empirically validated before.

Build environment reproducibility is also important for software preservation [33]. Previous works have observed [22, 31] that build systems, recipes, and tools, need to be preserved as much as source code [8] and binary executables. But preservation is less useful if, after having captured all of that, the build process cannot be replicated to obtain the intended result.

6 FUTURE PLANS

This study brings preliminary evidence that Nix allows specifying build environments that are reproducible both in space and time, and that most often, this enables rebuilding past software versions. We intend to complete this initial work by exploring more largely our dataset in order to understand the limits of this property: by rebuilding more Nixpkgs revisions, we will observe the evolution over time of package rebuildability. We have used Nix 2.6 to perform our experiments and relied on the fact that Hydra used diverse Nix versions over time to claim that Nix achieves build environment reproducibility with different Nix versions, but it would strengthen our results to make the local version of Nix vary as well. Since we collected logs of our local rebuilds and that Hydra's build logs are also available, we can analyze them to understand why some packages fail to rebuild, but also to measure how often the build logs are fully identical. We have argued that Nix reproducible environments enable rebuilding a component long after it was defined, but we have only tested running the build step, with all dependencies pulled from the Nix binary-cache. More failures are to be expected if we do not rely on a binary-cache, in particular when the program sources become unavailable. Because we are interested in the question of software preservation, we plan to evaluate the amount of packages that we are not able to build because of sources unavailability and the proportion we can salvage using sources archives like Software Heritage [8]. Finally, we are interested in the abilities of functional package managers for reproducible builds, for the impacts it can have on the security of the software supply chain [16]. We anticipate that the reproducibility of build environments enabled by Nix can help achieve reproducible builds, but the effect of other factors like compiler behaviors and quality of packaging are also to be considered.

Data availability. A full replication package for the experiments described in this paper is available from Zenodo [20] and archived on Software Heritage with SWHID `swh:1:rev:f513eee162ea28ab3066eb1c0aac57b80f16cc5c`.

REFERENCES

- [1] Pietro Abate, Roberto Di Cosmo, Louis Gesbert, Fabrice Le Fessant, Ralf Treinen, and Stefano Zacchiroli. 2015. Mining Component Repositories for Installability Issues. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*, Massimiliano Di Penta, Martin Pinzger, and Romain Robbes (Eds.). IEEE Computer Society, 24–33. <https://doi.org/10.1109/MSR.2015.10>
- [2] Carl Boettiger. 2015. An Introduction to Docker for Reproducible Research. *SIGOPS Oper. Syst. Rev.* 49, 1 (Jan 2015), 71–79. <https://doi.org/10.1145/2723872.2723882>
- [3] Jürgen Cito and Harald C. Gall. 2016. Using Docker Containers to Improve Reproducibility in Software Engineering Research. In *Proceedings of the 38th International Conference on Software Engineering Companion (Austin, Texas) (ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 906–907. <https://doi.org/10.1145/2889160.2891057>
- [4] Christian Couder. [n.d.]. Fully automated bisecting with "git bisect run" [LWN.net]. <https://lwn.net/Articles/317154/>
- [5] Ludovic Courtès. 2013. Functional Package Management with Guix. In *Proceedings of ELS 2013 - 6th European Lisp Symposium, Madrid, Spain, June 3-4, 2013*, Christian Queinnee and Manuel Serrano (Eds.). ELSAA, 4–14. <https://european-lisp-symposium.org/static/proceedings/2013.pdf#page=10>
- [6] Ludovic Courtès and Ricardo Wurmus. 2015. Reproducible and User-Controlled Software Environments in HPC with Guix. In *Euro-Par 2015: Parallel Processing Workshops - Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 9523)*, Sascha Hunold, Alexandru Costan, Domingo Giménez, Alexandru Iosup, Laura Ricci, María Engracia Gómez Requena, Vittorio Scarano, Ana Lucia Varbanescu, Stephen L. Scott, Stefan Lankes, Josef Weidendorfer, and Michael Alexander (Eds.). Springer, 579–591. https://doi.org/10.1007/978-3-319-27308-2_47
- [7] Julien Courtiel, Paul Dorbec, and Romain Lecoq. 2022. Theoretical Analysis of git bisect. In *LATIN 2022: Theoretical Informatics (Lecture Notes in Computer Science)*, Armando Castañeda and Francisco Rodríguez-Henriquer (Eds.). Springer International Publishing, Cham, 157–171. https://doi.org/10.1007/978-3-031-20624-5_10
- [8] Roberto Di Cosmo and Stefano Zacchiroli. 2017. Software Heritage: Why and How to Preserve Software Source Code. In *iPRES 2017 - 14th International Conference on Digital Preservation*. Kyoto, Japan, 1–10. <https://hal.science/hal-01590958>
- [9] Stephanie Dick and Daniel Volmar. 2018. DLL Hell: Software Dependencies, Failure, and the Maintenance of Microsoft Windows. *IEEE Annals of the History of Computing* 40, 4 (Oct. 2018), 28–51. <https://doi.org/10.1109/MAHC.2018.2877913> Conference Name: IEEE Annals of the History of Computing.
- [10] Eelco Dolstra. 2006. *The purely functional software deployment model*. Ph.D. Dissertation. s.n., S.l. OCLC: 71702886.
- [11] Eelco Dolstra, Andres Löb, and Nicolas Pierron. 2010. NixOS: A purely functional Linux distribution. *J. Funct. Program.* 20, 5-6 (2010), 577–615. <https://doi.org/10.1017/S0956796810000195>
- [12] Paul M. Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education. Google-Books-ID: PV9qfEdu9L0C.
- [13] Pronoy Goswami, Saksham Gupta, Zhiyuan Li, Na Meng, and Danfeng Daphne Yao. 2020. Investigating The Reproducibility of NPM Packages. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 677–681. <https://doi.org/10.1109/ICSM46990.2020.00071>
- [14] Hydra and nix-eval-jobs contributors. 2020–2023. nix-eval-jobs. <https://github.com/nix-community/nix-eval-jobs>
- [15] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and Evolution of Package Dependency Networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 102–112. <https://doi.org/10.1109/MSR.2017.55>
- [16] Chris Lamb and Stefano Zacchiroli. 2022. Reproducible Builds: Increasing the Integrity of Software Supply Chains. *IEEE Softw.* 39, 2 (2022), 62–70. <https://doi.org/10.1109/MS.2021.3073045>
- [17] Qingzhou Luo, Farah Hariri, Lamya Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 643–653. <https://doi.org/10.1145/2635868.2635920>
- [18] Julien Malka. 2023. fix recurseForDerivations evaluation in force-recurse mode by JulienMalka · Pull Request #206 · nix-community/nix-eval-jobs. <https://github.com/nix-community/nix-eval-jobs/pull/206>
- [19] Julien Malka. 2023. hydra-eval-jobs: fix jobs containing a dot being dropped by JulienMalka · Pull Request #1286 · NixOS/hydra. <https://github.com/NixOS/hydra/pull/1286>
- [20] Julien Malka. 2024. Replication package for: Reproducibility of Build Environments through Space and Time. <https://doi.org/10.5281/zenodo.10519820>
- [21] Dmitry Marakasov. 2016–2023. Repology, the packaging hub. <https://repology.org/>
- [22] Brian Matthews, Esther Conway, Jim Woodcock, Catherine Mary Jones, Juan Bicaregui, and Arif Shaon. 2009. Towards a Methodology for Software Preservation. In *Proceedings of the 6th International Conference on Digital Preservation, iPRES 2009, San Francisco, CA, USA, October 5-6, 2009*. <https://hdl.handle.net/11353/10.294040>
- [23] Mathias Meyer. 2014. Continuous Integration and Its Tools. *IEEE Software* 31, 3 (May 2014), 14–16. <https://doi.org/10.1109/MS.2014.58> Conference Name: IEEE Software.
- [24] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing dependency errors for Python build reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 439–451. <https://doi.org/10.1145/3460319.3464797>
- [25] Daniel Nüst, Vanessa Sochat, Ben Marwick, Stephen J. Eglon, Tim Head, Tony Hirst, and Benjamin D. Evans. 2020. Ten simple rules for writing Dockerfiles for reproducible data science. *PLoS Computational Biology* 16, 11 (Nov. 2020), e1008316. <https://doi.org/10.1371/journal.pcbi.1008316> Publisher: Public Library of Science.
- [26] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A Survey of Flaky Tests. *ACM Trans. Softw. Eng. Methodol.* 31, 1 (Oct. 2021), 17:1–17:74. <https://doi.org/10.1145/3476105>
- [27] Jeffrey M. Perkel. 2020. Challenge to scientists: does your ten-year-old code still run? *Nature* 584, 7822 (Aug. 2020), 656–658. <https://doi.org/10.1038/d41586-020-02462-7> Bandiera_abtest: a Cg_type: Technology Feature Number: 7822 Publisher: Nature Publishing Group Subject_term: Computational biology and bioinformatics, Computer science, Research data, Software.
- [28] Prins Pjotr, Jeeva Suresh, and Eelco Dolstra. 2008. Nix fixes dependency hell on all Linux distributions. <https://web.archive.org/web/20150708101023/http://archive09.linux.com/feature/155922>
- [29] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 345–355. <https://doi.org/10.1109/MSR.2017.54>
- [30] ReScience. 2020. Ten Years Reproducibility Challenge. <http://rescience.github.io/ten-years/>
- [31] Mahadev Satyanarayanan. 2018. Saving software from oblivion. *IEEE Spectrum* 55, 10 (Oct. 2018), 36–41. <https://doi.org/10.1109/MSPEC.2018.8482422> Conference Name: IEEE Spectrum.
- [32] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 724–734. <https://doi.org/10.1145/2568225.2568255>
- [33] Len Shustek. 2006. What Should We Collect to Preserve the History of Software? *IEEE Annals of the History of Computing* 28, 4 (Oct. 2006), 112–111. <https://doi.org/10.1109/MAHC.2006.78> Conference Name: IEEE Annals of the History of Computing.
- [34] César Soto-Valero, Nicolas Harrant, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the Maven ecosystem. *Empir Software Eng* 26, 3 (March 2021), 45. <https://doi.org/10.1007/s10664-020-09914-8>
- [35] Francesco Strozzi, Roel Janssen, Ricardo Wurmus, Michael R. Crusoe, George Githinji, Paolo Di Tommaso, Dominique Belhachemi, Steffen Möller, Geert Smart, Joep de Ligt, and Pjotr Prins. 2019. *Scalable Workflows and Reproducible Data Analysis for Genomics*. Springer New York, New York, NY, 723–745. https://doi.org/10.1007/978-1-4939-9074-0_24
- [36] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. 2020. An empirical characterization of bad practices in continuous integration. *Empir Software Eng* 25, 2 (March 2020), 1095–1135. <https://doi.org/10.1007/s10664-019-09785-8>
- [37] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. 2019. A large-scale empirical study of compiler errors in continuous integration. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 176–187. <https://doi.org/10.1145/3338906.3338917>