Docker does not Guarantee Reproducibility

Julien Malka
julien.malka@telecom-paris.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

The reproducibility of software environments is a critical concern in modern software engineering, with ramifications ranging from the effectiveness of collaboration workflows to software supply chain security and scientific reproducibility. Containerization technologies like Docker address this problem by encapsulating software environments into shareable filesystem snapshots known as images. While Docker is frequently cited in the literature as a tool that enables reproducibility in theory, the extent of its guarantees and limitations in practice remains under-explored.

In this work, we address this gap through two complementary approaches. First, we conduct a systematic literature review to examine how Docker is framed in scientific discourse on reproducibility and to identify documented best practices for writing Dockerfiles enabling reproducible image building. Then, we perform a large-scale empirical study of 5298 Docker builds collected from GitHub workflows. By rebuilding these images and comparing the results with their historical counterparts, we assess the real reproducibility of Docker images and evaluate the effectiveness of the best practices identified in the literature.

Keywords

Reproducibility, containers, Docker, Dockerfile, software development environment, software build environment

1 Introduction

The *reproducibility of build environments* [12, 25]—i.e., the ability to recreate the technical environment needed to develop and deploy a specific piece of software—is a fundamental aspect of modern software engineering, with significant implications for both developers and users. For developers, build environment reproducibility enables reliable recompilation of software from source code—even years after its initial creation—facilitating collaboration, long-term maintenance, and debugging. For end users, the ability to rebuild software from source can provide security guarantees, as it avoids the need of trusting opaque binaries built by third parties [23].

Yet, the growing complexity of software components and their dependencies [13], driven by composability and modular design, makes reproducing exact build environments increasingly challenging. This issue, which contributes to the broader *reproducibility crisis* [18, 33] issue, is well-documented in scientific computing [21, 42], where re-running old code to validate or extend prior work is a integral part of the methodology. Yet the problem extends beyond academia and touches all fields of software engineering.

Containerization technologies [6] allow software environments to be encapsulated and packaged as filesystem snapshots known as *container images*, usually in the OCI standard format [3], which can then be shared and executed on other systems. Docker [1] is a widely adopted open-source container runtime and build tool that popularized the use of build recipes called *Dockerfiles* to create container images. The straightforward syntax and developer-friendly tooling of Docker contributed to its wide adoption across the open source ecosystem, to the point that a 2020 study mining Dockerfiles from the World of Code archive found that over 1.9 million distinct repositories used them [15]. Docker supports a broad range of workflows in software engineering—including building, packaging, distributing, and deploying software—by making software environments transportable across systems. Thanks to this portability aspect, Docker is frequently recommended in both industry and academia for improving the reproducibility of computational environments [30].

However, Docker images are build artifacts, who suffer from common limitations for that type of software artifacts:

- they are large, binary artifacts that must be explicitly stored and archived, to avoid becoming unavailable over time [32];
- they are typically tied to specific hardware architectures, limiting their portability [7];
- they contain arbitrary binary artifacts that are difficult to audit and can represent a security risk [28, 31];
- modifying them to upgrade contained artifacts or change their behavior is impractical [31].

To address these limitations, it is essential to be able to *reliably rebuild a Docker image from its Dockerfile*, ensuring that the rebuilt image matches the original one. Different meanings of "matches" can be considered for this need.

When images can be rebuilt in a way that produces *bitwise-identical* copies (as in the *reproducible builds* approach [23]), trusting third-party distributors like public image registries become easier, strengthening the security of the software supply chain. Even when bitwise reproduction is not achievable, *functional equivalence*—where the rebuilt image contains the same software components, at the same versions, of the original image—remains an essential requirement. Failing it, discrepancies in image content may lead to observable differences when using the image.

Research questions. In this work, we study Docker as a tool to enable the reproducibility of software environments, with a particular focus on the *gap between theory and practice*: what is Docker *believed* to do in that respect versus what it *actually* does. Specifically, we address the following research questions:

RQ1: What are the claims and recommendations in the scientific literature on the reproducibility of Docker images?

RQ2: To what extent can Docker images built in the past be reproduced today from their Dockerfiles, to various degrees of equivalence (bitwise identical, functionally equivalent)?

RQ3: Are the recommendations present in the scientific literature about Docker image reproducibility effective?

Contributions. We make two primary contributions:

(1) We perform a **systematic literature review (SLR)** of scientific literature to identify how Docker is discussed in the context of reproducibility and map documented best practices for writing Dockerfiles that, allegedly, enable image reproducibility (RQ1).
(2) We conduct **a large-scale empirical study of Docker image reproducibility**. Leveraging a dataset of GitHub workflow runs, we reproduce and analyze 5298 historical Docker builds captured in October 2023. By comparing rebuilt images with their historic counterparts, we quantify the level of reproducibility of real-world Docker builds (RQ2). Then we empirically assess how the best practices for Docker reproducibility documented in the literature are used by developers, and whether they lead to improved reproducibility (RQ3).

Data availability statement: We make our code required to run our analysis available in the replication package [16].

2 Background

2.1 Software reproducibility

This article is concerned with the notion of *reproducibility* in software engineering. Reproducibility can take various meanings, and in this work, we explore several of them.

The strictest notion of reproducibility is *bitwise reproducibility*, a.k.a. *reproducible builds*. It is defined as the ability to reproduce the exact same binary artifact (bitwise identical) from a fixed source code and build environment. This definition has the advantage of being easy to verify, and it is thus useful both from a software supply chain security perspective [23], and from a scientific perspective [**?**], to show that the source code accompanying a scientific publication can be used to reproduce the results of the publication.

From a software engineering perspective, however, more relaxed definitions of reproducibility are just as useful. For instance, *functional reproducibility* focuses on the behavior of the software rather than its exact binary representation. Therefore, when comparing two environments, we will consider them functionally equivalent if they contain the same software components, at the same versions. What to consider as "same versions" can then be discussed. If we want bug-by-bug reproducibility, we should use strictly the same versions. But if we are only interested in compatibility, we can relax this constraint to same minor version, or even same major version.

Finally, rebuildability is an even looser notion of reproducibility, requiring only that the build process still runs successfully. In Docker's context, this means we can still build an image from the Dockerfile, even if the result is not bitwise identical or functionally equivalent to the original. This remains useful for reuse, though it offers fewer guarantees about the relationship between the resulting software environment and the historical one.

2.2 Containers, Docker images and Dockerfiles

A *container* is a lightweight, isolated execution environment that runs a process with its own filesystem, networking, and process namespace, while sharing the host operating system kernel [6]. Containers are designed for fast startup, resource efficiency, and high portability. A Docker [1] image is a standardized, self-contained package that includes all the components needed to run an application in a container: executable binaries, system libraries, configuration files, etc. Internally, it is an immutable filesystem snapshot made of a superposition of read-only *layers*. Each layer encodes a set of filesystem changes such as file additions, modifications, or deletions. This layering model enables efficient storage and distribution: shared layers can be cached and reused across images. Container images follow the Open Container Initiative (OCI) image specification [3] and may be produced by various build tools, but the most widely adopted one is the built-in Docker builder.

The Docker builder takes as input a build recipe called a *Dockerfile*. Dockerfiles specify how to construct an image from a sequence of instructions, such as selecting a base image, copying files, installing packages, and defining environment variables. For example:

```
FROM python:3.13-slim
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . /app
CMD ["python", "/app/main.py"]
```

This Dockerfile creates an image from a minimal Python base, installs project dependencies, copies the application code, and defines the startup command. Dockerfiles are imperative and order-sensitive: each instruction creates a new image layer that may affect later build steps. Each layer is content-addressed by a cryptographic hash of its contents and metadata. To improve reproducibility, the Docker builder supports the `SOURCE_DATE_EPOCH` variable, which standardizes timestamps in generated files and metadata, helping produce identical hashes across builds.

2.3 GitHub actions

GitHub Actions [8] is a continuous integration and deployment (CI/CD) service natively integrated with the GitHub collaborative development platform. It enables developers to automate workflows triggered by repository events—such as code pushes or pull requests—by executing a series of commands in virtual machines (called *runners*) provided by GitHub. Workflows are specified using YAML files stored directly in the repository. Each workflow consists of one or more *jobs*, in turn composed of ordered *steps*. Steps run sequentially within the same job environment, allowing state propagation—e.g., via the filesystem or environment variables—making step chaining key for composing logic in CI pipelines. Each step may invoke a shell command or call a reusable, modular unit of functionality known as a *GitHub Action*. Actions are typically provided by other GitHub repositories, are versioned, and configurable. They encapsulate common tasks (such as checking out code, setting up a programming language environment, or uploading artifacts), promoting reuse across repositories and workflows.

An example of a GitHub Action is `docker/build-push-action`, a widely used action maintained by Docker and designed to build container images from Dockerfiles and optionally push them to a registry. It supports numerous parameters to customize the Docker build, which are translated to flags passed to the `docker build` command. Here is an excerpt of a workflow using this action:

```yaml
steps:
  - uses: actions/checkout@v4
  - uses: docker/build-push-action@v6
    with:
      context: .
      push: true
      tags: myname/myapp:latest
```

In this example, the official GitHub checkout action is used to clone the repository, then the `docker/build-push-action` builds a Docker image from a Dockerfile at the root of the repository. This workflow could be further simplified because the Docker action supports doing the repository checkout by itself.

3 Systematic literature review

To answer RQ1 we conducted a systematic literature review (SLR), adhering to established best practices as outlined in the *Empirical Standards for Software Engineering* [35]. This SLR is a scoping and critical review [36], whose primary objective is to extract and categorize the academic discourse on Docker reproducibility, with a focus on identifying expressed beliefs, and recommendations for writing Dockerfiles that enable reproducible image building. SLR outcomes will serve as foundations for an empirical study, where we evaluate to which extent these beliefs and recommendations are founded (RQ2, RQ3). We start by refining RQ1 into the following sub-questions:

RQ1.a: What are the expectations, implicit or explicit, present in the scientific literature on the reproducibility guarantees provided by Docker?

RQ1.b: What are the common causes, documented in the scientific literature, of non-reproducibility of Docker images?

RQ1.c: What are the recommendations, put forward in the literature, to maximize the reproducibility of Docker images?

3.1 Methodology

3.1.1 Search protocol. We began with a prototyping phase to define our search strategy. We performed initial queries across standard scientific databases including SCOPUS, IEEE Xplore, the ACM Digital Library, and Springer. However, because the search is limited to title and abstract on several of these databases, our queries returned very few results. This suggests that only few publications address Docker reproducibility as their main contribution, and that a full content search was necessary to identify relevant discussions occurring in broader contexts. For that purpose, we switched to Google Scholar, which support full-text search queries. This also allowed us to find more types of scientific literature such as tutorials, theses, and preprints. Based on our research questions, we settled on the following query terms:

(1) "reproducible science" + dockerfile, yielding 149 results;
(2) "reproducible build environment" + docker, 13 results;
(3) reproducibility + smells + dockerfile, 150 results;
(4) reproducibility + dockerfile + linter, 220 results.

3.1.2 Inclusion and exclusion criteria. Following the search phase, we removed 82 duplicate entries, leaving 450 unique articles. We then applied a three-stage filtering process based on the title, abstract, and full-text content. At each phase, the first and last authors devised the inclusion and exclusion criteria by processing a subset of entries together, discussing potential criteria until each entry was included or excluded without ambiguity. The criteria were intentionally designed to exclude articles that were clearly irrelevant, while retaining those with uncertainty for further evaluation in subsequent filtering phases. Once criteria were stabilized, first and last author shared the workload of checking individual articles.

Table 1: Filtering criteria used to select articles for the systematic literature review on Docker reproducibility.

Stage	Type	Criterion
Title	Inclusion	**IT1:** Title contains keywords related to reproducibility **IT2:** Title contains keywords related to Docker or containers **IT3:** Title contains keywords related to infrastructure as code
	Exclusion	**EC1:** Title does not contain any keyword mentioned in inclusion criteria
Abstract	Inclusion	**IA1:** Contribution explicitly related to Docker or reproducibility best practices
	Exclusion	**EA1:** Contribution not related to Docker **EA2:** Only uses Docker without reflection on Docker as a tool
Full text	Inclusion	**IC1:** Contains recommendations on Dockerfile best practices **IC2:** Recommends or advises against using Docker for reproducibility **IC3:** Addresses smells or bad practices in Dockerfiles
	Exclusion	**EC1:** Article full text not accessible to authors **EC2:** No references only anecdotal ones to Dockerfiles **EC3:** Tool contribution without reproducibility or quality focus

Table 1 summarizes the criteria applied at each stage. During title screening, we retained articles whose titles mentioned reproducibility, Docker or containers, or infrastructure as code. This step eliminated 271 articles. Next, we reviewed the abstracts of the remaining articles. We retained only those that explicitly discussed Docker as part of their contribution or addressed best practices related to reproducibility in software engineering or scientific workflows. Articles that merely used Docker (e.g., to provide a replication package) without reflecting on its role as a tool were excluded. After this phase, 83 articles remained. Finally, during the full-text screening phase, articles were included if they offered actionable recommendations for writing Dockerfiles, discussed the use of Docker for reproducibility, or discussed Dockerfile smells and anti-patterns. We excluded articles that were inaccessible, lacked any relevant mention of Dockerfiles, or presented Dockerfile generating tools without any focus on reproducibility or quality aspects.

3.1.3 Backward snowballing. We performed backward snowballing by examining the related work section (or the introduction, when there was no related work section, or the full text for literature reviews) of all papers selected after the abstract filtering phase.

References cited in these sections were considered for snowballing based on their title and their description in the paper that cites them. This process yielded 26 additional papers, 13 of which met our inclusion criteria. Additionally, we included the official Docker documentation on best practices for building containers, as it was frequently cited in the selected literature. We ended up with a total of 60 papers to analyze.

3.1.4 Content analysis. We adopted a qualitative content analysis approach inspired by Charmaz constructive grounded theory [9], structured in two main phases. In the *initial coding phase*, the first and last authors shared the workload of reading the full texts of the selected articles and identified excerpts relevant to the research questions. During this open-ended process, excerpts were assigned short, descriptive codes that captured key ideas, practices, or concerns related to Docker and reproducibility. During this phase, when multiple versions of a paper existed (4 cases with two versions), they reviewed one in depth, then screened the other to check for any additional relevant content. In the *focused coding phase*, all authors together examined the obtained codes to uncover patterns and similarities across the dataset. They discussed and agreed on a new set of refined codes to form a structured taxonomy of academic recommendations and beliefs related to Dockerfile quality and the reproducibility of containerized software environments. In this phase, we also sorted the papers into two main classes: software engineering papers, where Docker is the *object of study*, and scientific computing papers, where Docker is used as a *tool* to provide reproducibility for scientific workflows. We obtained (after removal of the 4 duplicates) 31 software engineering papers (denoted in the following SE1 to SE31) and 18 scientific computing papers (denoted SC1 to SC18). The other papers are denoted O1 (the official Docker recommendations) to O7. The correspondence between identifiers and full references is available in our replication package [16].

3.2 Results

Our results in this section are mainly qualitative, but we start by providing some quantitative statistics about the papers included in the SLR. The 31 software engineering papers (SE) include 7 journal articles, 18 conference papers, 4 master theses, and 2 preprints. The most represented venues are Mining Software Repositories (MSR) with 4 papers, followed by the International Conference on Software Engineering (ICSE), the International Conference on Software Maintenance and Evolution (ICSME), and the Empirical Software Engineering journal (EMSE) with 3 papers each. Publication years range from 2017 to 2025 with 4 to 5 papers each year except for 2017 and 2019 (1 paper each) and 2025 with 2 papers so far.

The 18 scientific computing papers (SC) include 14 journal papers, 3 conference papers, and 1 textbook, covering a wide range of research areas, including bioinformatics and life sciences (5 papers), computer science (4 papers), psychology (3 papers), and astronomy (1 paper). The remaining 5 papers are general discussions on reproducibility in scientific computing. The publication years range from 2015 to 2024, with 2 to 3 papers each year except for 2015, 2018, and 2024 (only 1 paper each) and 2020 (no paper). The oldest paper in our collection (SC3) has been cited more than 1000 times.

The 7 other papers (O) include 3 practitioners books, 1 magazine article, 1 conference paper, 1 master thesis, and 1 web page.

The results of our content analysis is a taxonomy of beliefs and recommendations that we detail graphically in our replication package. In the following, we summarize the findings that are most relevant to our research questions.

3.2.1 RQ1.a: Docker reproducibility expectations. Among the main SLR results, we find that the scientific literature contains a wide range of beliefs and expectations about Docker reproducibility, many of which *contradict each other*. On the one side, Docker is often framed as a tool for reproducibility, with bold claims such as "containerization with Docker guarantees full computational reproducibility" (SC1) or "containers cannot be matched when it comes to enabling reproducibility in a lightweight and portable manner" (SC8), in particular in the context of scientific computing, where reproducing the exact environment used to produce results is viewed as essential: "full computational reproducibility is only achieved if the software versions used originally are precisely documented." (SC1). These claims are however rather vague in terms of *what* exactly enables reproducibility (is it the distribution of Docker images? of Dockerfiles?).

Other quotes from these and other papers however clarify the importance of the Dockerfile for reproducibility and transparency: "it is [...] essential that the process of creating and building containers themselves is transparent and reproducible" (SC13). Yet, the expectations are sometimes that reproduction of a Docker image from a Dockerfile is straightforward: "These containers can be replicated from a single blueprint specified by a text file known as a Dockerfile." (SE12) and that Dockerfiles are precise and unambiguous: "There is little possibility of the kind of holes or imprecision in such a script that so frequently cause difficulty in manually implemented documentation of dependencies" (SC3); "Dockerfiles are declarative definitions of an environment that aim to enable reproducible builds of the container." (SE4). This is however debated by other authors: "Dockerfiles are often not easy to follow, and in many cases, do not clarify which specific packages are being deployed" (SE15).

Sometimes, sharing a Docker image or Dockerfile is presented as equivalent: "There are two ways to share a Docker image; either by sharing the Dockerfile that creates the image or by sharing the image itself [...]. While both ways guarantee a replicable computational environment, sharing the Dockerfile is more transparent and more space-saving." (SC1); "By using Docker containers, developers need only share a Docker image or Dockerfile, which guarantees the environment will be the same" (SC18).

On the other hand, this is contrasted by quotes that highlight the limitations of Docker for long-term reproducibility, which should in principle reach at least a decade (SC9). While it is claimed that: "One of the core promises of Docker is that images are stable environments that can be downloaded and run even years after their creation." (SE3), this is also contrasted by the lack of guarantees regarding long-term reproducibility of Docker images: "Ideally, containers that we built years ago should rebuild seamlessly, but this is not necessarily the case" (SC16). Even when the Dockerfile still builds, it may not produce an identical environment: "Dockerfile does not guarantee the same build every time" (SC8); "it is highly unlikely that over time you will be able to recreate [the image] precisely from the accompanying Dockerfile" (SC16).

Another reason why sharing a Dockerfile is considered important is that it allows to reproduce the build environment with controlled variation, which can be important for both auditing and building upon past scientific experiments: "As some of these changes may indeed resolve valid bugs or earlier problems [...], it will often be insufficient to demonstrate that results can be reproduced when using the original versions" (SC3); "The layer of abstraction that a Dockerfile provides makes it easier to change [...] versions or experiment parameters later on." (SC2).

3.2.2 RQ1.b: Causes of non-reproducibility. The most commonly documented cause of non-reproducibility (which may manifest in terms of a Dockerfile that does not build, or builds but produces a different image) is related to changes in the external environment: "temporal failures that emerge over time due to dynamic changes in external dependencies, such as updates to base images, third-party libraries, or environmental settings—occurring without any modifications to the Dockerfile itself" (SE13). In particular, it can be due to changes in the dependencies: "the absence of a concrete version can lead to the usage of a version which is not compatible with the other components of the container, thus to failed builds, or failures hitting surface only at container execution" (SE4). Dependencies, such as the base image, can also disappear: "images that are not pulled by anyone from Docker Hub for extended periods of time get purged, and Dockerfiles are not guaranteed to build indefinitely" (SC8). Finally, the Dockerfile in itself may not be sufficient to reproduce the build, if other important aspects such as the build context are not correctly preserved: "many of them are not reproducible for builds as [...] their build context files are missing" (SE12).

3.2.3 RQ1.c: Recommendations for Dockerfile reproducibility. Most of the recommendations discussed in the literature (particularly in software engineering papers) are not directly related to reproducibility, but rather to image size, build time, and security. Some of these recommendations (such as avoiding bloat by not installing unnecessary dependencies) can however impact reproducibility. More generally, using linters to detect Dockerfile smells [24] is a common recommendation, supported by the belief that "Dockerfiles [...] quality may affect the reproducibility [...] of the resulting image." (SE1). The most commonly used tool in many publications, but also for practitioners, is `hadolint` [2]: "Hadolint is currently the reference tool" (SE24) and official images are often presented as a reference for quality: "The official status of a repository can be seen as a quality label" (SE3).

Given the main cause for non-reproducibility described above, the most common recommendation to improve reproducibility is to pin dependencies to specific versions, whether they are installed packages, base images, or dependencies pulled from Git repositories: "If a copy is done via git clone or equivalent, a specific commit or a tagged git version should be specified, never a branch only." (SC13). Yet, there is a trade-off between pinning dependencies and the ability to get security updates: "pinning package versions requires continuous maintenance to keep versions up to date" (SE24). This leads to contradictory recommendations in the literature: e.g., for the base image, some recommend pinning by tag or by digest, while others recommend the use of mutable tags (latest, or major versions tags that still receive updates): "mutating tags for major releases, which bring regular updates in, may be desirable and in other cases stricter reproducibility is desired" (SE25). Similarly, some recommend running `apt-get update` before installing packages, while others recommend *against* it.

This is why the choice of which best practices to follow depends on the objectives, and practices can vary widely: "These practices might be somewhat different from the practices of generic software developers in that researchers often need to focus on transparency and understandability rather than performance considerations." (SC16). Nevertheless, images are also viewed as opaque and difficult to audit, so, for software supply chain security, pinning dependencies can also be seen as a good practice (O1), which can then be balanced by using tools to update pinned dependencies regularly.

Among other contradictory recommendations, we found that some authors recommend reusing base images to avoid duplicating work, when others avoid non-official base images, and prefer reusing and adapting their Dockerfiles instead. A third route is reusing base images, but after ensuring that they are rebuildable from their Dockerfile, and saving the Dockerfile in the repository.

Best practices for reproducibility also cover including in version control all the files copied in the image, avoiding the use of `ARG` which can make the build process less transparent, documenting the build process, and avoiding to build parts of the image outside the Docker build process: "It should be noted that building the software outside containers and only including the final binary in the container is considered a bad practice." (SE25).

Finally, some authors recommend using other recipes than a Dockerfile and associated tools, be it Maven for Java images (O5, but not specifically for reproducibility purposes) or functional package managers, such as Nix or Guix (SC7, SC17, O3).

4 Experimental Docker image reproducibility

In this section, we describe our methodology for studying Docker image reproducibility empirically, and answer RQ2 and RQ3.

4.1 Methodology

4.1.1 Creation of the dataset. In order to study the reproducibility of image-building from Dockerfiles, one needs access to historically built Docker images as well as corresponding Dockerfile, source code, and build parameters used to generate them—serving as a ground truth for reproducibility analysis. While several existing datasets contain Dockerfiles, Docker images, or both, to the best of our knowledge, none provides the precise association between images and the complete build configuration (including repository commit) that produced them.

To address this gap, we leverage the *GHALogs* dataset [27], which captures over 500 000 GitHub Actions workflow runs, along with logs and metadata, collected across 25 000 software repositories in October 2023. We focus specifically on workflows that use the `docker/build-push-action`—the official GitHub Action for building and publishing Docker images to repositories. This action's metadata contain all relevant build parameters (e.g., Dockerfile path, build context, arguments, and target stage) necessary to locally reproduce the image build. Besides, when images are pushed to a repository, we can analyze traces left in the execution logs.

To create our dataset, we apply a series of filtering and cleaning steps to the raw GHALogs data. We first discard all workflow runs

that do not invoke the Docker build-push action. To avoid potential side effects from prior steps that may alter the build context in ways difficult to reproduce experimentally, we only allow runs with a small list of actions that can be executed before the build-push step, which we know how to reproduce the effect of, or which do not have an effect on the build: `action/checkout`, and Docker's `login-action`, `metadata-action`, `setup-buildx-action` and `setup-qemu-action`. After these filtering steps, we obtain a cleaned dataset of 6622 relevant workflow runs, of which 5298 completed successfully in July–October 2023.

Collection of historical images. To form our ground truth, we parse the build logs included in GHALogs, extracting both the destination repository where each image was pushed and its `SHA256` digest, which uniquely identifies the image content. This approach allows us to retrieve the original push location of 3620 historical images. The remaining cases mostly correspond to workflows that built images without pushing them to a registry. Among these 3620 images, we successfully retrieved 1541; the remaining 2079 were no longer accessible at the time of our analysis. This number is not unexpected, as images are routinely deleted from registries.

4.1.2 Empirical study. We locally rebuild all images that historically built successfully. All relevant parameters from the `build-push` and `setup-buildx` actions such as build arguments, context, target, are passed to the `docker build` command. However, variations in time, Docker versions, host operating system, and finally hardware are inevitably introduced, which we consider to be acceptable when testing for reproducibility. While historical builds have been performed by GitHub runners and in most part on hardware provided by GitHub, we use our own pipeline, on our own `x86_64-linux` hardware, to facilitate customization and analysis. Each image is built in isolation within a fresh virtual machine to prevent cross-contamination between builds and to eliminate interference from the host environment. We use the return code of the `docker build` command to classify the build as a success or a failure, and export the resulting image in case of success. In any case, we keep the complete build log to identify transient failures (like disk space exhaustion) and re-run them, and to allow future analysis.

Bitwise reproducibility analysis. We perform a file-by-file comparison of each historical image against its rebuilt version. For each pair of historical image matched with the rebuilt one, we walk through the image file tree and compare the content of each file. We count the proportion of files with different content, or files that are missing from one of the images.

Functional reproducibility analysis. To assess whether rebuilt images provide the same software environment as their historical counterpart, we analyze the packages installed in the images and their versions. We use Trivy [4], an open source tool that inspects container images and extracts package metadata from package managers to produce a software bill of material. Trivy supports a wide range of system and ecosystem-specific package managers, allowing us to achieve a good coverage of analyzed images. Still, some projects strip all package management metadata from their container images to minimize bloat which prevents the analysis. We run Trivy on both the historical and rebuilt images, when both exists, setting a timeout of 25 minutes. We collect the extracted package lists for comparison. This step works for 993 of the 1120 images for which both versions were available. Using the extracted information, for each pair of images, we compute three metrics, comparing image content at the package level. From the strictest, to the most permissive, we compute the proportion of packages:

(1) with a different version string, or missing in one image;
(2) with a different minor version, or missing in one image;
(3) with a different major version, or missing in one image.

When checking for identical minor or major versions, we split version numbers by components and keep only the first or the first two components.

Reproducibility rules detection. We examine the Dockerfiles in our dataset for reproducibility related issues and best practice violations identified by the SLR. Following common recommendations, we use `hadolint` as the primary tool to detect Dockerfile quality issues that may impact reproducibility. The first and last authors reviewed the full set of `hadolint` rules, selecting those that matched recommendations extracted during SLR and are related to reproducibility, either directly, or indirectly through the objective of avoiding image bloat. We also implement additional rules not directly covered by `hadolint` and easily identifiable at the Dockerfile or context level. The resulting set of rules is summarized in Table 2.

Statistical analysis. We perform a statistical analysis to understand whether there is a statistical dependency between the Dockerfile quality recommendations identified in the literature and our measured reproducibility outcomes for the corresponding Docker images. Since the recommendations may be correlated to one another, testing each one independently for statistical significance would risk overlooking confounding effects. To account for potential correlations, we perform regressions, using the presence or absence of each individual rule as input variables and the reproducibility metric under study as the output variable. This approach allows us to address two questions:

(1) Is there a statistically significant relationship between violations of reproducibility-related recommendations and the observed reproducibility of Docker images?
(2) If so, which individual recommendations are significantly associated with more reproducible outcomes?

Comparison with projects from the official Docker library. SLR results highlighted two common beliefs in academic discourse: (1) Dockerfile quality impacts reproducibility, and (2) Dockerfiles of official images are authored by Docker experts, and thus are of particularly high quality. To evaluate these claims, we extracted Dockerfiles from the `docker-library/official-images` repository, which tracks all versions of the Dockerfiles used to build official library images. We took a snapshot of this repository from October 2023, matching the timeframe of the collection of the rest of our Dockerfiles. We then rebuild the collected Dockerfiles using the same experimental pipeline as the community-sourced ones, allowing us to directly compare the reproducibility of expert-authored Dockerfiles with those mined from the wider ecosystem.

Table 2: Reproducibility recommendations extracted from the literature and tested against our Dockerfiles using Hadolint rules (DL) or custom rules (CR).

Rule code	Recommendation that triggers the rule violation when not respected
Pin Dependencies	
DL3006	Tag images versions explicitly.
DL3007	Do not use `latest`.
DL3008	Pin versions in `apt-get install`.
DL3013	Pin versions in `pip`.
DL3016	Pin versions in `npm`.
DL3018	Pin versions in `apk add`.
DL3028	Pin versions in `gem install`.
DL3033	Pin versions in `yum install`.
DL3035	Do not use `zypper dist-upgrade`.
DL3037	Pin versions in `zypper install`.
DL3041	Pin versions in `dnf install`.
Avoid Temporary Files	
DL3009	Delete the `apt-get` lists after installing.
DL3010	Use `ADD` for extracting archives into an image.
DL3032	Run `yum clean all` after installing.
DL3036	Run `zypper clean` after installing.
DL3040	Run `dnf clean` after installing.
DL3042	Avoid cache directory with `pip install --no-cache-dir`
DL3060	Run `yarn cache clean` after installing.
CR01	Use a `.dockerignore` file to avoid useless files.
CR02	Use multi-staged builds.
Avoid Unneeded Dependencies	
DL3015	Avoid additional packages by specifying `--no-install-recommends`.
Fail Early	
DL4006	Set the `SHELL` option `-o pipefail` before `RUN` with a pipe in it
Use official images	
CR03	Use images from the official docker library

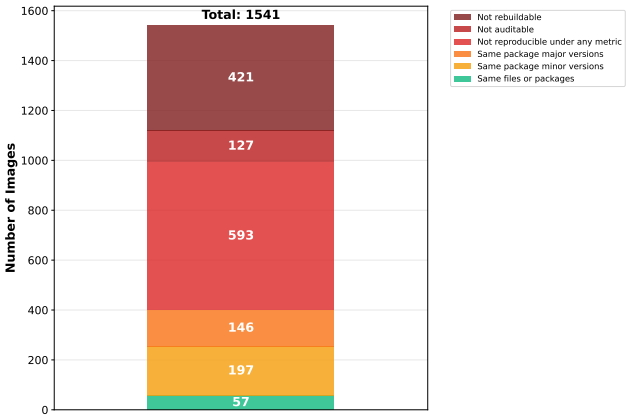

Figure 1: Number of images fully verifying each reproducibility metric, with the 4 reproducible under our "same file" policy merged with "same packages" for lisibility.

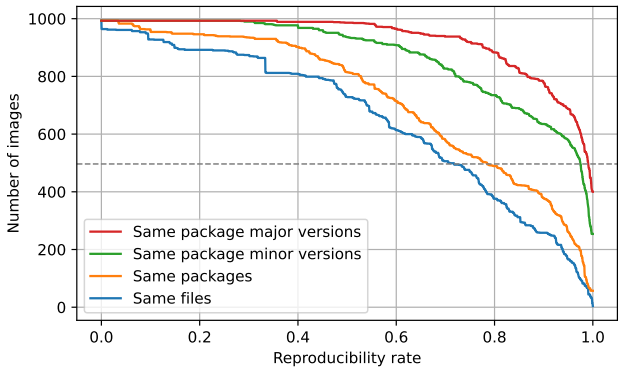

Figure 2: Number of images achieving a given reproducibility rate on each of our four metrics.

4.2 Results

We present our experimental results below, organized by research question.

4.2.1 RQ2: Levels of reproducibility of built images. We report on the level of reproducibility measured on the Docker images in our dataset. We consider an image to be fully reproducible under a certain metric when all *packages* (resp. *files* for bitwise reproducibility) are reproducible. Figure 1 summarizes the number of images that are fully reproducible under each metric. Additionally, we discuss the proportion of packages reproducible for each metric, starting with the strictest definition of reproducibility and progressively relaxing it towards more functional metrics.

Bitwise reproducibility. Although the default Docker builder supports the `SOURCE_DATE_EPOCH` environment variable since February 2023 to set the date metadata of the image layers, none of the workflows in our dataset use it, and as a consequence none of the images we built have the same output hash as their historical counterpart. When looking at files content only, we find that only 4 images have all their files bitwise reproducible with their historical version, and half of the images in our dataset have more than 28.2% of files with differing hashes.

Functional reproducibility. When looking at images through the package lens, we still observe very low reproducibility rate with at least 21.5% of packages changing version between the original and rebuilt images in half of the cases. Comparing package versions this way, only 57 of the compared images achieve full reproducibility.

Although when comparing minor versions, we get a drastically lower proportion of non-reproducible packages—half the images have less than 3% differing packages—still only a minority of 254 out of 993 images are reproducible under this prism. This number increases slightly when considering only major version changes, where 400 images are reproducible, with a median of 1% packages changing major version. Figure 2 shows the number of images achieving a given reproducibility rate for each metric.

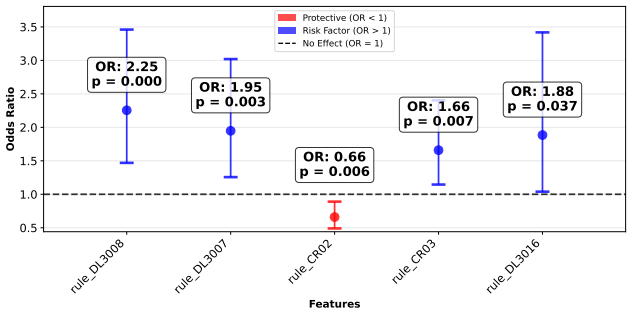

Figure 3: Significant rules for rebuildability: blue variables are rules that decrease rebuildability when violated, red ones that increase it. Rules `DL3008`, `DL3007` and `DL3016` are pinning rules.

Rebuildability. Finally, the most minimal expectation for reproducibility is being able to rebuild an image at all. While all 5298 jobs that we tried to build did produce an image in July–October 2023, we find that less than two years later only 3836 are still building successfully, hence resulting in a 72% rebuildability rate.

4.2.2 RQ3: Effectiveness of recommendations for reproducibility. We perform regression analyses to understand whether there is a statistically significant relationship between the reproducibility recommendations we extracted from the literature and the various reproducibility outcomes observed in our study. For each regression, the input variables are binary indicators representing for each rule whether it is violated. The output variable is either the proportion of unreproducible packages—used in linear regressions for package-level reproducibility—or the build outcome (success or failure), modeled using logistic regression for rebuildability. Before performing the regressions, we filter out rules that appear in less that 1% of our examples, because they may lead to overfitting. We observe a p-value lower than 0.05 for each of our models, showing a statistically significant dependency between the recommendations studied and our metrics. Furthermore, for each metric, some of the rules tested also have a statistically significant effect. However, the fit of our models, with R^2 respectively of 0.04 for rebuildability, 0.20 for file reproducibility, 0.13 for exact package version reproducibility, 0.13 for minor version reproducibility, and 0.09 for major version reproducibility shows a poor explanation power of the models on variability of the reproducibility rates. Furthermore, interpretation of the individual rules coefficients is also hazardous as, for all but the rebuildability metrics, the rules fitted have very weak effects, explaining only a few percentage points of the variability. In that context, we even see in some cases counter-intuitive results with some pinning rules appearing to contribute negatively to the reproducibility rate for some metrics, but with very small effect size (below 3%). This suggests that these models are only measuring a weak signal and that there might be confounding factors at play, like the evolution of the base image used or of the system package manager. We show in Figure 3 the four significative rules for our rebuildability model, and all other regressions results are available in our replication package [16].

Reproducibility of the Docker official images. Another angle to understand if quality differences in Dockerfiles may have an impact on reproducibility is to test Dockerfiles expected to be of high quality. That is what we do by rebuilding the 121 Dockerfiles from 2023 extracted from Docker official images, successfully rebuilding 106, that is 88%. This 16 percentage points increase in rebuildability rate compared to the Dockerfiles extracted from the wider ecosystem seems to confirm that the quality of the Dockerfile may have an impact on rebuildability, but that is is not the only factor at play and that there might be factors influencing rebuildability that cannot be controlled by the Dockerfile alone.

5 Discussion

5.1 Takeaways

In this work, we have looked at the question of Docker images reproducibility through the prism of five different reproducibility levels tied to varying expectations and needs. Our empirical findings show that **Docker provides no guarantees for any of these reproducibility levels**. This can break some assumptions held by the scientific community, as highlighted by our systematic literature review (SLR). Our results show that sharing a Dockerfile is not equivalent to sharing a built Docker image, and that with time the content of the image may change, increasing the chances of impeding the functional behavior of the image when the software environment changes too much. Even more symptomatic, although not necessarily worse, images may even stop building altogether when external resources go missing or the build environment varies significantly.

For software engineering purposes, these concerns compounds with another one: the inability of Docker to generate bitwise identical images **forces users to blindly trust the images they are downloading to be faithfully generated** from its documented sources. In a context where high profile attacks on the software supply chain become more common [22], this lack of proof of correspondence between source code and generated image places Docker at a disadvantage compared to other means of software distribution that provide such guarantees, for example some Linux distributions like Debian or Nixpkgs. Although we have shown that there is a statistically significant effect of some of the recommendations found in the literature about image reproducibility, it is to be put in perspective with the modest size of the aforementioned effects. This suggests that **following best practices to write Dockerfiles alone helps, but is not a silver bullet** for achieving image reproducibility in Docker-based workflows.

Our findings should not be seen as a call to abandon neither containerization technologies nor Docker. These tools fulfill important roles in software engineering: they provide portability, simplify deployment, and enable execution environment isolation. However, we believe that users and projects relying on Dockerfiles to build container images should critically assess their Docker usage in light of our results. If image reproducibility is an important concern for their workflows, our findings suggest they may need to adjust their practices—or even reconsider their tooling—to better align with their reproducibility goals. Users who cannot easily switch to alternative tools should consider adopting the best practices we

experimentally validated as contributing positively to reproducibility, such as pinning dependencies whenever possible. However, they should not do so with the expectation that this will *guarantee* reproducibility of their images.

Our SLR also highlights **alternative approaches—such as the use of functional package managers**—whose reproducibility guarantees, both for rebuildability and bitwise reproducibility, are backed by recent experimental results [25, 26]. Notably, functional package managers remain usable with the Docker runtime, as they can generate OCI-compliant images while preserving the reproducibility benefits of their model.

In reviewing the literature, we were struck by the diversity—and at times, **contradiction—of the beliefs and recommendations** we encountered, occasionally even within the same article. We also observed that some papers used vague or imprecise terminology, which often led to questionable or unsupported claims. Furthermore, and this justifies our use of the term "beliefs," we found that few of the assertions promoting Docker as a tool for reproducibility or for use in scientific workflows were actually supported by empirical evidence or grounded in research. This is not caused by a disconnect between research fields to the point that empirical research on Docker would not be read or taken into account by scientists in other disciplines, but rather by the **extreme scarcity of software engineering work on this topic**. This is concerning, as it suggests that technical decisions shaping scientific practice are being made without input from software engineering domain experts. Such a disconnect carries the risk of contributing to systemic issues like the reproducibility crisis. It is our opinion that it is the responsibility of software engineering researchers to support the broader scientific community by sharing their expertise and helping guide the adoption of sound technical practices. It is in this spirit that we undertook this work.

5.2 Threats to validity

We adopt the terminology of Runeson et al. [39] in this section.

5.2.1 Construct validity. A first concern is that the SLR could have missed some relevant papers, in particular by the choice of search keywords. This concern is mitigated by the use of Google Scholar full-text search to cover all types of papers, complemented by the backward snowballing. Furthermore, according to Ralph et al. [36], this concern is not as relevant for critical reviews, where a subset of papers can be sufficient to highlight problems in the analyzed area. In our case, the papers we analyzed were indeed sufficient to reveal many contradictory statements and unfounded beliefs.

Another concern regarding the SLR is that we may have misinterpreted some of the papers, or that the quotes we extracted do not represent the authors' intent. However, it is important to note that even if these quotes may not perfectly represent the *authors' view*, they carry weight as they can influence what other researchers and practitioners will believe about Docker reproducibility based on what is *written*. Therefore, it is important to be able to highlight when some of these statements are misleading.

Regarding our empirical study, the main risk is that our convenience sample for testing reproducibility from the GHALogs dataset is not representative of the wider population of Dockerfiles and Docker images. Indeed, we cannot claim that the figures that we obtained would be the same on different samples, and we even show this by comparing to a gold set of Dockerfiles from the official Docker library. Nonetheless, the exact reproducibility rates are less important than our main finding: that reproducibility of image building is not guaranteed by Docker out of the box.

5.2.2 Internal validity. When relating smells in Dockerfiles to the reproducibility of the images built from them, we control for the possibility that smells are correlated with each other, by performing a regression analysis that takes into account all smells at the same time. However, it is very likely that some confounding factors that cannot be observed directly are at play. Therefore, we make no claim of causal effects in our results, and we discourage the reader to misinterpret them as showing that some recommendations are proved to be effective. Besides, while some recommendations are indeed correlated with the reproducibility of the output in our regression results, the effect sizes are quite small, which means that these recommendations alone are not sufficient to achieve the reproducibility objective.

5.2.3 External validity. As already noted, we do not claim that our figures would generalize to other sets of Dockerfiles. Furthermore, even when using the same sample, these results cannot generalize in time. We performed our reproducibility study less than two years after the original builds, and it is likely that we would have obtained higher reproducibility rates if we had performed the study earlier, and that one will obtain lower rates in the future.

5.2.4 Reliability. For the SLR, we used constructive grounded theory [9] to derive a taxonomy of beliefs and recommendations. This methodology does not adhere to the positivist view of reliability, and therefore other researchers could have derived a different taxonomy. The way to judge this research is therefore not whether it can be repeated yielding identical results, but whether the results are useful and grounded in the data we collected. To allow closer inspection of our research process and results, we provide detailed information about it in our replication package [16].

For the empirical study, our replication package [16] includes all code used for the analysis. It can be used to replicate our study, subject to the time-related caveat described above. However, due to data size, we do not provide all intermediate outputs, such as built images.

6 Related work

6.1 Docker studies

In our work, we have collected data from Docker builds, Dockerfiles, and Docker images, and we have run an experiment consisting in rebuilding Docker images from their Dockerfiles. There are many studies in empirical software engineering that focus on Docker, and which have done one or more of these steps. To the best of our knowledge, however, none of them has done all of them together, and few have focused on reproducibility.

Collecting Dockerfiles. Many studies have mined Dockerfiles from public repositories (most often from GitHub) to study their quality, usage, and evolution. Among the earliest work on the topic, Cito et al. [10] collected 70 000 Dockerfiles from GitHub to study their quality, using `hadolint` to detect smells. This is also the case

of Henkel et al. who collected 178 000 Dockerfiles from GitHub [19], and later used them as an empirical basis for the design of a new linter (Binnacle) [20], or Zhou et al. [47] who similarly collected Dockerfiles to derive rules for a new linter (DRIVE). We initially considered running these two linters in addition to `hadolint` in our empirical study, but we found that none of them was properly packaged for reuse (no installation or usage instructions). More recently, Durieux collected Dockerfiles and manually annotated a subset to serve as a ground truth for evaluating linting and repair tools [14]. His own tool, `Parfum` is however focused on smells related to image size. Finally, the largest study of Dockerfiles to date is by Eng et al., where they used World of Code to analyze 9 million Dockerfiles, and study their evolution over time [15].

Collecting Docker images. Other authors have collected Docker images from public registries (generally Docker Hub). For instance, Zerouali et al. [45] collect Docker images from Docker Hub to analyze technical lag. The notion of technical lag [17] is close to the metrics that we use for evaluating functional reproducibility, as they share the idea of measuring and aggregating a drift in versions of packages present in an image. The main difference is that technical lag is a measure of the distance to an ideal version (usually the latest), while our functional reproducibility metrics are a comparison between two images, one historical and one rebuilt.

Sometimes, Docker images are associated with (GitHub) repositories containing Dockerfiles. That is the case in the study by Lin et al. [24] that investigates at the same time characteristics of Docker images (such as their size) and Dockerfiles (such as the smells they contain). None of these datasets, however, provide precise links between a specific version (repository commit) of a Dockerfile and the built image, nor the exact build parameters, which are essential for reproducibility analysis.

Collecting Docker build results. Some studies collect Docker build data, usually from the Docker Hub automated builds, and not from GitHub workflows like we did. Wu et al. [43] collected Docker build statuses, but their study focuses on how often build failures occur, how long it takes to fix them, and the evolution of these metrics over time. In later work [44], they also look at build durations. Similar to us, they correlate characteristics of Dockerfiles (including `hadolint` smells) with build duration. However, they do not try to rebuild images. The same can be said of a study by Zhang et al. [46] that also correlates build duration and Dockerfile characteristics.

Rebuilding Docker images. A few studies before ours have attempted to rebuild Docker images from Dockerfiles. Cito et al. [10] did this for a random sample of 560 Dockerfiles from their dataset mentioned above, and observed a 34% failure rate. Similar to us, they observed more build successes when Dockerfiles used image pinning. However, in their study, the original build status of the Dockerfile had not been collected. Therefore the failed builds could also have been from Dockerfiles that had never built successfully in the first place, or by not using the right build arguments. Besides, each Dockerfile smell was correlated with build success rate independently of the others, which is not as reliable to draw conclusions, because smells can be correlated with each other.

Shabani et al. [40] also rebuilt Dockerfiles but, in their case, with the goal of studying the flakiness of Docker builds. They started by building a large number of Dockerfiles (about 18 000), and observe a 55% failure rate (with the same limitations as Cito et al.). However, they then proceeded to rebuild the Dockerfiles that built successfully every week for 9 months, and observe the failures. Interestingly, while the aim of the study was to study flakiness, which is normally understood as a transient failure, they report a large proportion of "permanent flakiness", i.e., Dockerfiles that used to build and then stop building after some time. Although their focus was not reproducibility, and the authors did not measure the semantic drift of rebuilt images, their study is the closest to ours.

Finally, Zhu et al. [48] also rebuilt Dockerfiles to evaluate the optimization they apply to them, but they do not report on the initial failure rate of the collected Dockerfiles.

6.2 Reproducibility studies

The so-called "reproducibility crisis" [18, 33] in empirical sciences is largely documented and has led to many studies focusing on the reproducibility of scientific experiments. We do not aim to review these works here. Instead, we focus specifically on the software engineering literature on reproducibility.

Reproducibility is an important concern in software engineering for security [23], collaboration workflows, maintenance [38], software preservation [11], and applications to scientific computing.

In the context of the latter, Hassan et al. [18] performed a systematic literature review on *reproducibility debt*, which they define as technical debt affecting scientific software and the resulting ability to reproduce scientific results. While they do not specifically focus on the reproducibility of software environments, several of the causes of reproducibility debt that they report are connected to this aspect (e.g., missing dependencies, issues with software versions).

Reproducibility of build environments has been investigated by Malka et al. [25] in the context of the Nix functional package manager. A later study by the same authors [26] used an experimental approach similar to ours, consisting in re-running builds for which a historical truth is still available. However, their focus is on bitwise reproducibility, which they show can be achieved at a large scale using Nix. In our context, most Docker images were far from bitwise reproducible, so we had to resort to more relaxed definitions of reproducibility. Other relaxed definitions have been investigated before, e.g., by Pöll et al. [34], with the concept of *accountable builds*, in which differences between builds that can be explained are tolerated, or by Sharma et al. [41], who apply a canonicalization step to build artifacts of unreproducible builds.

There are more experimental studies of reproducibility [5, 37]. However, these studies most often rebuild the same software twice, rather than comparing a build today, with a historical build.

Finally, while containerization raises the question of whether images can be built reproducibly, as we investigated here, it can also be seen as a tool for reproducible builds, as shown by Navarro et al. [29] with their *reproducible containers*. Fundamentally, containerization is tied to the concept of sandboxing, which is also what functional package managers use to improve reproducibility.

7 Conclusion

In this work, we investigated the reproducibility of Docker images built from Dockerfiles. Through a survey of the literature,

we uncovered beliefs expressed in the scientific discourse about Docker images reproducibility—including that Dockerfiles do provide reproducibility—and recommendations associated to image reproducibility. We empirically tested these claims by rebuilding 5298 historical images from 2023 GitHub actions workflows. Our results show that Docker does not guarantee reproducibility under any tested definition, nor is there a "silver bullet" set of rules for writing Dockerfiles yielding reproducible images.

References

[1] [n. d.]. Docker. https://www.docker.com/. Website, accessed 2025-07-10.

[2] [n. d.]. Haskell Dockerfile Linter. https://github.com/hadolint/hadolint. Website, accessed 2025-07-17.

[3] [n. d.]. Open Container Initiative - Open Container Initiative. https://opencontainers.org/

[4] [n. d.]. Trivy: The all-in-one open source security scanner. https://trivy.dev/. Website, accessed 2025-07-17.

[5] Giacomo Benedetti, Oreofe Solarin, Courtney Miller, Greg Tystahl, William Enck, Christian Kästner, Alexandros Kapravelos, Alessio Merlo, and Luca Verderame. 2025. An Empirical Study on Reproducible Packaging in Open-Source Ecosystems.

[6] Ouafa Bentaleb, Adam S. Z. Belloum, Abderrazak Sebaa, and Aouaouche El-Maouhab. 2022. Containerization technologies: taxonomies, applications and challenges. *J. Supercomput.* 78, 1 (2022), 1144–1181. doi:10.1007/S11227-021-03914-1

[7] Carl Boettiger. 2015. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review* 49, 1 (Jan. 2015), 71–79. doi:10.1145/2723872.2723882

[8] Chaminda Chandrasekara and Pushpa Herath. 2021. Introduction to GitHub actions. In *Hands-on GitHub actions: implement CI/CD with GitHub action workflows for your applications*. Springer, 1–8.

[9] Kathy Charmaz. 2014. *Constructing Grounded Theory* (2nd édition ed.). SAGE Publications Ltd, London ; Thousand Oaks, Calif.

[10] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. 2017. An Empirical Analysis of the Docker Container Ecosystem on GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 323–333. doi:10.1109/MSR.2017.67

[11] Ludovic Courtès, Timothy Sample, Stefano Zacchiroli, and Simon Tournier. 2024. Source Code Archiving to the Rescue of Reproducible Deployment. In *Proceedings of the 2nd ACM Conference on Reproducibility and Replicability, ACM REP 2024, Rennes, France, June 18-20, 2024*. ACM. doi:10.1145/3641525.3663622

[12] 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. doi:10.1007/978-3-319-27308-2_47

[13] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Softw. Engg.* 24, 1 (Feb. 2019), 381–416. doi:10.1007/s10664-017-9589-y

[14] Thomas Durieux. 2024. Empirical Study of the Docker Smells Impact on the Image Size. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, Lisbon Portugal, 1–12. doi:10.1145/3597503.3639143

[15] Kalvin Eng and Abram Hindle. 2021. Revisiting Dockerfiles in Open Source Software Over Time. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 449–459. doi:10.1109/MSR52588.2021.00057

[17] Jesus M. Gonzalez-Barahona, Paul Sherwood, Gregorio Robles, and Daniel Izquierdo. 2017. Technical lag in software compilations: Measuring how outdated a software deployment is. In *Open Source Systems: Towards Robust Practices: 13th IFIP WG 2.13 International Conference, OSS 2017, Buenos Aires, Argentina, May 22-23, 2017, Proceedings 13*. Springer International Publishing, 182–192. https://library.oapen.org/bitstream/handle/20.500.12657/27754/1002251.pdf?sequence=1#page=188

[18] Zara Hassan, Christoph Treude, Michael Norrish, Graham Williams, and Alex Potanin. 2025. Characterising reproducibility debt in scientific software: A systematic literature review. *Journal of Systems and Software* 222 (April 2025), 112327. doi:10.1016/j.jss.2024.112327

[19] Jordan Henkel, Christian Bird, Shuvendu K. Lahiri, and Thomas Reps. 2020. A Dataset of Dockerfiles. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*. Association for Computing Machinery, New York, NY, USA, 528–532. doi:10.1145/3379597.3387498

[20] Jordan Henkel, Christian Bird, Shuvendu K. Lahiri, and Thomas Reps. 2020. Learning from, understanding, and supporting DevOps artifacts for docker. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 38–49. doi:10.1145/3377811.3380406

[21] Peter Ivie and Douglas Thain. 2018. Reproducibility in Scientific Computing. *ACM Comput. Surv.* 51, 3 (2018), 63:1–63:36. doi:10.1145/3186266

[22] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2023. SoK: Taxonomy of Attacks on Open-Source Software Supply Chains. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 1509–1526. doi:10.1109/SP46215.2023.10179304

[23] Chris Lamb and Stefano Zacchiroli. 2022. Reproducible Builds: Increasing the Integrity of Software Supply Chains. *IEEE Software* 39, 2 (March 2022), 62–70. doi:10.1109/MS.2021.3073045 Conference Name: IEEE Software.

[24] Changyuan Lin, Sarah Nadi, and Hamzeh Khazaei. 2020. A large-scale data set and an empirical study of docker images hosted on docker hub. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 371–381. https://ieeexplore.ieee.org/abstract/document/9240654/

[16] Julien Malka. 2025. Docker does Not Guarantee Reproducibility - Replication Package. https://gitlab.telecom-paris.fr/julien.malka/docker-does-not-guarantee-reproducibility

[25] Julien Malka, Stefano Zacchiroli, and Théo Zimmermann. 2024. Reproducibility of Build Environments through Space and Time. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, NIER@ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 97–101. doi:10.1145/3639476.3639767

[26] Julien Malka, Stefano Zacchiroli, and Théo Zimmermann. 2025. Does Functional Package Management Enable Reproducible Builds at Scale? Yes.. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. 775–787. doi:10.1109/MSR66628.2025.00115 ISSN: 2574-3864.

[27] Florent Moriconi, Thomas Durieux, Jean-Rémy Falleri, Raphaël Troncy, and Aurélien Francillon. 2025. GHALogs: Large-Scale Dataset of GitHub Actions Runs. IEEE Computer Society, 669–673. doi:10.1109/MSR66628.2025.00104

[28] Dave Morris, S. Voutsinas, Nigel C. Hambly, and Robert G. Mann. 2017. Use of Docker for deployment and testing of astronomy software. *Astronomy and computing* 20 (2017), 105–119. https://www.sciencedirect.com/science/article/pii/S2213133717300161?casa_token=OMnoq8fYgncAAAAA:tmDtpPxiNUvpxsWu_YCtXsnVAQJX_aE3ViTu2vVyyY45JfsOSR9RrztEN8Qou5Ff65czjpqT Publisher: Elsevier.

[29] Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. 2020. Reproducible Containers *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 167–182. doi:10.1145/3373376.3378519

[30] Daniel Nüst, Vanessa Sochat, Ben Marwick, Stephen J. Eglen, 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. doi:10.1371/journal.pcbi.1008316 Publisher: Public Library of Science.

[31] Maximiliano Osorio, Carlos Buil-Aranda, Idafen Santana-Perez, and Daniel Garijo. 2022. DockerPedia: A Knowledge Graph of Software Images and Their Metadata. *International Journal of Software Engineering and Knowledge Engineering* 32, 01 (Jan. 2022), 71–89. doi:10.1142/S0218194022500036

[32] Aaron Peikert and Andreas M. Brandmaier. 2021. A reproducible data analysis workflow with R Markdown, Git, Make, and Docker. *Quantitative and Computational Methods in Behavioral Sciences* (2021), 1–27. https://qcmb.psychopen.eu/index.php/qcmb/article/view/3763

[33] Jeffrey M. Perkel. 2020. Challenge to scientists: does your ten-year-old code still run? *Nature* 584, 7822 (Aug. 2020), 656–658. doi: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.

[34] Manuel Pöll and Michael Roland. 2021. Analyzing the Reproducibility of System Image Builds from the Android Open Source Project. (2021). https://www.digidow.eu/publications/2021-poell-tr-reproducibilityaospsystemimages/Poell_2021_ReproducibilityAOSPSystemImages.pdf

[35] Paul Ralph, Nauman bin Ali, Sebastian Baltes, Domenico Bianculli, Jessica Diaz, Yvonne Dittrich, Neil Ernst, Michael Felderer, Robert Feldt, Antonio Filieri, Breno Bernard Nicolau de França, Carlo Alberto Furia, Greg Gay, Nicolas Gold, Daniel Graziotin, Pinjia He, Rashina Hoda, Natalia Juristo, Barbara Kitchenham, Valentina Lenarduzzi, Jorge Martínez, Jorge Melegati, Daniel Mendez, Tim Menzies, Jefferson Molleri, Dietmar Pfahl, Romain Robbes, Daniel Russo, Nyyti Saarimäki, Federica Sarro, Davide Taibi, Janet Siegmund, Diomidis Spinellis, Miroslaw Staron, Klaas Stol, Margaret-Anne Storey, Davide Taibi, Damian Tamburri, Marco Torchiano, Christoph Treude, Burak Turhan, Xiaofeng Wang, and Sira Vegas. 2021. Empirical Standards for Software Engineering Research. doi:10.48550/arXiv.2010.03525 arXiv:2010.03525 [cs].

[36] Paul Ralph and Sebastian Baltes. 2022. Paving the way for mature secondary research: the seven types of literature review. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1632–1636. doi:10.1145/3540250.3560877

[37] Georges Aaron Randrianaina, Djamel Eddine Khelladi, Olivier Zendra, and Mathieu Acher. 2024. Options Matter: Documenting and Fixing Non-Reproducible Builds in Highly-Configurable Systems. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 654–664. https://ieeexplore.ieee.org/abstract/document/10555868/

[38] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. 2018. Automated localization for unreproducible builds. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 71–81. doi:10.1145/3180155.3180224

[39] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (April 2009), 131–164. doi:10.1007/s10664-008-9102-8 Publisher: Springer Science and Business Media LLC.

[40] Taha Shabani, Noor Nashid, Parsa Alian, and Ali Mesbah. 2025. Dockerfile Flakiness: Characterization and Repair. doi:10.48550/arXiv.2408.05379 arXiv:2408.05379 [cs].

[41] Aman Sharma, Benoit Baudry, and Martin Monperrus. 2025. Canonicalization for Unreproducible Builds in Java. doi:10.48550/arXiv.2504.21679 arXiv:2504.21679 [cs].

[42] Victoria Stodden, Randall J. LeVeque, and Ian Mitchell. 2012. Reproducible Research for Scientific Computing: Tools and Strategies for Changing the Culture. *Comput. Sci. Eng.* 14, 4 (2012), 13–17. doi:10.1109/MCSE.2012.38

[] Gael Vila, Emmanuel Medernach, Ines Gonzalez Pepe, Axel Bonnet, Yohan Chatelain, Michael Sdika, Tristan Glatard, and Sorina Camarasu Pop. 2024. The Impact of Hardware Variability on Applications Packaged with Docker and Guix: a Case Study in Neuroimaging. In *Proceedings of the 2nd ACM Conference on Reproducibility and Replicability (ACM REP '24)*. Association for Computing Machinery, New York, NY, USA, 75–84. doi:10.1145/3641525.3663626

[43] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. 2020. An Empirical Study of Build Failures in the Docker Context. In *2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)*. 76–80. doi:10.1145/3379597.3387483 ISSN: 2574-3864.

[44] Yiwen Wu, Yang Zhang, Kele Xu, Tao Wang, and Huaimin Wang. 2022. Understanding and Predicting Docker Build Duration: An Empirical Study of Containerized Workflow of OSS Projects. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Rochester MI USA, 1–13. doi:10.1145/3551349.3556940

[45] Ahmed Zerouali, Tom Mens, Alexandre Decan, Jesus Gonzalez-Barahona, and Gregorio Robles. 2021. A multi-dimensional analysis of technical lag in Debian-based Docker images. *Empirical Software Engineering* 26, 2 (March 2021), 19. doi:10.1007/s10664-020-09908-6

[46] Yang Zhang, Gang Yin, Tao Wang, Yue Yu, and Huaimin Wang. 2018. An insight into the impact of dockerfile evolutionary trajectories on quality and latency. In *2018 ieee 42nd annual computer software and applications conference (compsac)*, Vol. 1. IEEE, 138–143. https://ieeexplore.ieee.org/abstract/document/8377650/?casa_token=e6jmpx3Et2IAAAAA:-f4a7FmybSY5TkvL8P8UO66u_4Hv-mlacfuyVkeDs72Sb42iqeUhZTOF3qisSloh7WCylTE

[47] Yu Zhou, Weilin Zhan, Zi Li, Tingting Han, Taolue Chen, and Harald Gall. 2023. DRIVE: Dockerfile Rule Mining and Violation Detection. *ACM Trans. Softw. Eng. Methodol.* 33, 2 (Dec. 2023), 30:1–30:23. doi:10.1145/3617173

[48] Zhiling Zhu, Tieming Chen, Chengwei Liu, Han Liu, Qijie Song, Zhengzi Xu, and Yang Liu. 2025. Doctor: Optimizing Container Rebuild Efficiency by Instruction Re-Orchestration. (April 2025). doi:10.48550/arXiv.2504.01742 arXiv:2504.01742 [cs].