A Catalog of Transformations to Remove Dockerfile Smells

Eduardo Barros
Federal University of Alagoas, Maceió, Brazil

Márcio Ribeiro
Federal University of Alagoas, Maceió, Brazil

Luana Martins
University of Salerno, Fisciano, Italy

Valeria Pontillo
Gran Sasso Science Institute L'Aquila, Italy

Fabio Palomba
University of Salerno, Fisciano, Italy

Baldoino Fonseca
Federal University of Alagoas, Maceió, Brazil

Rohit Gheyi
Federal University of Campina Grande, Brazil

Ivan Machado
Federal University of Bahia, Salvador, Brazil

Abstract

Background: Infrastructure-as-Code (IaC) has become a standard in software deployment, with Dockerfiles among its most prevalent formats. However, Dockerfiles often exhibit *Dockerfile smells*, violations of best practices that can harm reproducibility, build time, image size, and security. Although prior work has proposed detectors, fixes, and refactoring catalogs, existing knowledge remains mostly descriptive or illustrative, lacking explicit transformation rules with applicability conditions and behavior-preservation assumptions under stated preconditions. *Aims:* This work aims to provide a catalog of formalized transformation rules for removing Dockerfile smells and to evaluate how practitioners perceive the resulting Dockerfile. *Method:* We formalized rules for ten common Dockerfile smells and conducted a mixed-method empirical study with 59 participants, including 39 survey respondents and 20 interviewees, most of whom were experienced industry practitioners. *Results:* Participants generally preferred the resulting Dockerfiles, which achieved an overall mean acceptance rate of 76%. Eight of the ten rules were accepted by a majority of participants, with the highest acceptance rate reaching 95%. Participants strongly favored transformations that improved stability, security, and build time, but were less favorable toward transformations that reduced readability or made debugging harder. *Conclusions:* Our findings suggest that formalized Dockerfile transformations are a promising basis for refactoring support, but their adoption depends on balancing technical quality improvements with developers' practical expectations.

2012 ACM Subject Classification Software and its engineering → Empirical software validation; Software and its engineering → Maintaining software; Software and its engineering → Software defect analysis; Software and its engineering → Software evolution

Keywords and phrases Docker, Dockerfile, Refactoring, Catalog

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

In today's software development, applications are expected to be portable, scalable, and reproducible across execution environments [4]. At the same time, development processes increasingly rely on short and continuous release cycles [14, 7]. Containerization supports these requirements by packaging an application with its dependencies and execution environment into an isolated, reproducible unit. Among containerization technologies, Docker has become one of the most widely adopted solutions [20], relying on Dockerfiles to specify how container images are built [5].

Although Dockerfiles make execution environments explicit, writing them correctly is not trivial. Developers must define base images, dependencies, environment variables, files, and commands while following best practices related to maintainability, reproducibility,

performance, and security [9, 28, 36, 37]. When these practices are violated, Dockerfiles may contain recurring quality problems, known as *Dockerfile smells* [35]. Such smells can negatively affect image size, build time, reproducibility, and security. For example, if a Dockerfile does not specify a non-root user, the application runs as root inside the container, increasing the impact of any exploit, especially when access is granted to mounted volumes, sensitive files, or permissive capabilities. Likewise, using unpinned or outdated base images can introduce vulnerable dependencies or cause builds to resolve to inconsistent versions.

Prior research has substantially advanced our understanding of Dockerfile quality. Existing studies have investigated the prevalence of Dockerfile smells, their evolution, and the extent to which developers fix them [5, 35, 28]. Other studies have proposed taxonomies of Docker-specific technical debt categories and refactorings [18, 17]. However, these refactorings are still mostly proposed at a descriptive or illustrative level. As a result, it remains unclear how such refactorings can be systematically applied as concrete transformation rules, and whether developers perceive the resulting Dockerfiles as useful, readable, and acceptable in practice.

In this paper, we address this gap by presenting a catalog of formalized transformation rules that remove ten common Dockerfile smells. The transformations were derived from prior studies on Dockerfile smells and Docker-specific refactorings [18, 17, 27]. We selected transformations that address different quality concerns, including readability, maintainability, build efficiency, image size, reproducibility, and security. Each transformation is specified using source and target patterns, preconditions, and meta-variables, making the rules more actionable than high-level refactoring descriptions.

To validate this catalog, we conducted a mixed-method empirical study involving 59 participants: 39 survey respondents and 20 interviewees. Our results indicate a general preference for the formalized transformations. Overall, participants preferred the transformed Dockerfiles in 76% of the cases. Six transformations received very high acceptance rates, ranging from 83% to 95%; three received rates between 59% and 77%; and only one, *Extract RUN Instructions*, was rejected by most participants, with an acceptance rate of 25%. Participants strongly favored transformations that enhanced stability, security, and build caching efficiency. However, we also observed a tension between technical optimization and human readability [12]: developers sometimes rejected transformations when they increased cognitive load, reduced readability, or made debugging more difficult.

In summary, this paper makes the following contributions: (i) A catalog of formalized transformation rules for ten Dockerfile smells; (ii) An empirical evaluation of practitioners' perceptions of these transformations; and (iii) A qualitative analysis of the reasons why practitioners accept or reject Dockerfile transformations.

2 Background and Related Work

This section describes the background and the related work we build upon.

2.1 Background

Infrastructure-as-Code (IaC) treats infrastructure and deployment configuration as software artifacts that can be versioned, reviewed, tested, and maintained [31]. In the context of containerization, Docker is a platform that supports the packaging and execution of applications in isolated environments called containers. A container is created from a Docker image, which bundles the application together with the dependencies and configuration required for its execution.

Docker images are defined through Dockerfiles, composed of instructions written in a Domain Specific Language (DSL). Each instruction consists of a Docker-specific keyword (e.g., RUN) and one or more arguments. A Dockerfile can extend a base image, i.e., an existing Docker image that provides an initial filesystem and configuration for the new image.

During the build process, Docker executes the instructions in the Dockerfile to construct the final image. Docker images are composed of stacked layers, each corresponding to the result of executing one Dockerfile instruction. Docker also uses a layer caching mechanism, which allows previously built layers to be reused in subsequent builds of the same Dockerfile when the corresponding instructions and inputs have not changed.

Docker images are stored and distributed through registries, which may be publicly accessible or private with restricted access. Docker Hub is a widely used public registry maintained by Docker Inc. Images in registries are usually associated with tags that identify specific versions or variants, allowing developers to refer to different image versions when building or deploying containerized applications.

2.2 Related Work

The study of Infrastructure-as-Code (IaC) quality has gained significant attention in recent years. As a foundational work, Sharma et al. [31] analyzed 4,621 Puppet repositories to detect implementation and design configuration smells, establishing a basis for understanding quality issues in configuration code. In the specific context of containerization, Cito et al. [5] conducted an exploratory empirical study of the Docker container ecosystem on GitHub, analyzing over 70,000 Dockerfiles. They found that 29% of quality issues arise from missing version pinning and observed that developers frequently rely on heavy-weight operating systems as base images.

Subsequent work further characterized the prevalence and evolution of Dockerfile smells. Wu et al. [35] defined Dockerfile smells as Dockerfile instructions that violate recommended best practices and may negatively affect Dockerfile quality. They distinguished between two categories: DL-smells, which violate Dockerfile best practices, and SC-smells, which violate basic shell-script practices. Their empirical study of 6,334 GitHub projects showed that Dockerfile smells are highly prevalent, with nearly 84% of the analyzed projects containing smells, most of them being DL-smells.

Beyond smell detection, researchers have investigated whether Dockerfile smells are relevant to developers and whether they are willing to accept fixes. Rosa et al. [28] evaluated Hadolint smells from the perspective of expert Dockerfile developers and found that experts prioritize only a subset of the evaluated smells. They also proposed a ranked catalog containing 26 additional Dockerfile smells reported by the community. In a complementary study, Rosa et al. [27] analyzed smell-removing commits in 53,456 Dockerfiles and submitted pull requests with smell fixes to developers. Their results indicate that developers pay more attention to fixes related to performance, such as image size and build time, and are willing to accept fixes for common Dockerfile smells.

Researchers have also investigated Docker-specific refactoring practices. Ksontini et al. [18] proposed a taxonomy of generic and Docker-specific refactorings based on the analysis of 68 Docker projects, introducing 24 refactorings for Docker artifacts and associating them with 7 technical debt categories. A later study expanded this taxonomy to 41 refactoring mechanisms and 9 technical debt categories [17]. Tool support has also started to emerge. Ksontini et al. [16] introduced DRMiner, a tool that identifies and analyzes Dockerfile refactorings to identify refactoring candidates. More recently, Ksontini et al. [19] explored the potential of automating Dockerfile refactoring using Large Language Models via In-Context

Learning, reporting improvements in image size and build duration.

Although prior studies provide solid evidence on Dockerfile smells and refactoring practices, the knowledge remains largely descriptive, typically expressed through names, definitions, or examples rather than explicit operational rules. This limitation is particularly relevant as recent studies have shown that quality issues in specialized software artifacts evolve throughout the software lifecycle and require more systematic support for quality management and remediation [26]. Our work addresses this gap by formalizing selected smell fixes as transformation rules and empirically evaluating their perceived quality. This empirical step is crucial, as even formally correct transformations may not be adopted if developers perceive them as less readable, harder to debug, or misaligned with their practices.

3 Catalog of Formalized Transformations

This section presents our catalog of formalized transformations for Dockerfile smells. Each transformation is defined as an explicit rule that maps an original Dockerfile structure to a revised one, along with the metavariables used and the preconditions under which the transformation is intended to preserve behavior.

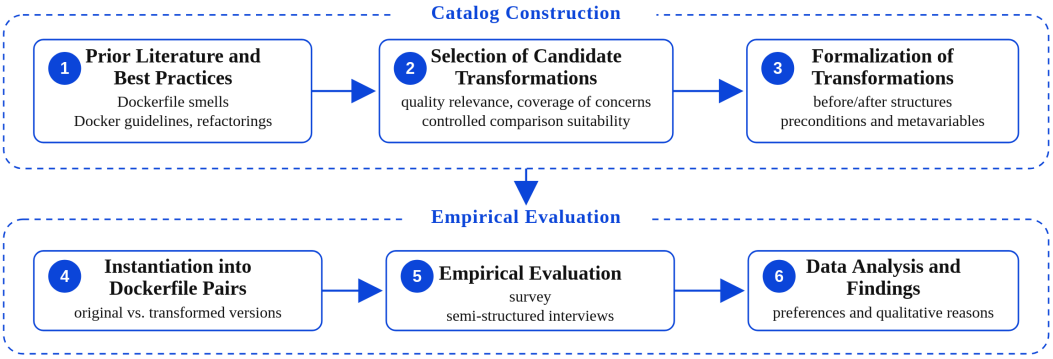


Figure 1 Overview of the research design, from the construction of the transformation catalog to its empirical evaluation.

Figure 1 summarizes our workflow. The first phase focuses on constructing the catalog: starting from Dockerfile smells, refactorings, and best practices reported in prior work, we selected transformations based on relevance, coverage of quality concerns, and suitability for controlled comparison. Each transformation was then formalized as an abstract rule comprising a source structure, a target structure, preconditions, and metavariables, as described in Section 3.2. The second phase concerns the empirical evaluation. We instantiated each rule into concrete before-and-after Dockerfile snippets and assessed them with practitioners through a survey and semi-structured interviews.

3.1 Selection of Transformations

We selected transformations from prior Dockerfile smell and refactoring literature based on four criteria: (i) they address a quality issue identified in earlier studies or official Docker best practices; (ii) they can be expressed as behavior-preserving changes under explicit applicability conditions; (iii) they admit a compact before/after representation suitable for controlled evaluation in surveys and interviews; and (iv) collectively, they cover a range of quality

concerns, including readability, maintainability, build efficiency, image size, reproducibility, and security.

The selection process involved reviewing candidate smells and refactorings from existing catalogs [18, 17, 27, 28, 35] and grouping them by the underlying change they prescribe. When multiple sources described similar practices, we consolidated them into a single transformation. For instance, recommendations to avoid untagged or `latest` base images were unified under *Explicit Version Pinning*, while refactorings that replace generic base images with more appropriate ones were captured as two variants of *Update Base Image*.

3.2 Formalization Procedure

Once we selected the transformations, we formalized them using a template-based representation proposed by Soares et al. [33]. In their work, refactorings are described through definitions and transformation templates composed of two main parts: a left-hand side (LHS), which describes the original smelly structure, and a right-hand side (RHS), which describes the refactored structure. When necessary, these templates are complemented by applicability conditions and by a discussion of the expected quality improvements.

We adapted this representation to the Dockerfile domain by specifying each transformation as an abstract rule over Dockerfile structures. Unlike test-code refactorings [21], Dockerfile transformations require domain-specific elements, such as image names, build stages, shell commands, paths, and dependency lists.

Each rule comprises four elements. The *left* structure captures the Dockerfile pattern where the smell or suboptimal practice occurs, while the *right* structure defines its transformed counterpart. The *preconditions* specify the applicability conditions under which the transformation preserves intended behavior, and the *metavariables* abstract over concrete elements such as base image names, commands, paths, stages, and dependency lists.

3.3 Catalog

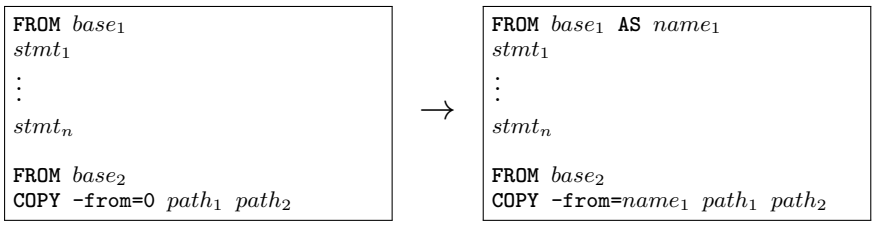
Table 1 summarizes the ten formalized transformations in this study, including the targeted smell and the technical debt category defined by Ksontini et al. [17]. Each transformation is presented using a compact rule format. We first briefly describe the quality problem addressed by the transformation and then present its formalization as a before-and-after pattern [33, 32, 1, 23]. The left-hand side represents the original Dockerfile structure affected by the smell, while the right-hand side represents the transformed structure. Preconditions define the assumptions required to preserve the intended behavior [24, 6, 25], and metavariables abstract over concrete Dockerfile elements such as image names, commands, paths, stages, and package lists.

3.3.1 T1. Multi-stage Build Naming

This transformation replaces index-based references to build stages with semantic aliases, reducing coupling between `COPY` instructions and the order of stages. It is based on the *Add or rename Image alias* refactoring [18].

Table 1 Overview of the formalized Dockerfile transformations.

ID	Transformation	Targeted smell	Tech. Debt Category
T1	Multi-stage Build Naming	Index-based stage reference	Understandability; maintainability
T2	Layer Optimization	Multiple consecutive `RUN` layers	Image size; build efficiency
T3	Update Base Image (Avoid Heavy)	Heavy general-purpose base image	Image size; security
T4	Update Base Image (Avoid Manual Install)	Manual installation of dependencies already available in existing images	Build time; maintainability
T5	Avoid Root	Container executed as root	Security
T6	Sort Instructions	Cache-invalidating instruction order	Build time
T7	Explicit Version Pinning	Missing or `latest` base image tag	Reproducibility; stability
T8	Extract Stage	Single-stage build containing build-time dependencies	Image size; modularity; security
T9	Replace `ADD` with `COPY`	Use of `ADD` when only local copying is needed	Readability; predictability
T10	Extract `RUN` Instructions	Complex inline shell command sequence	Modularity; maintainability

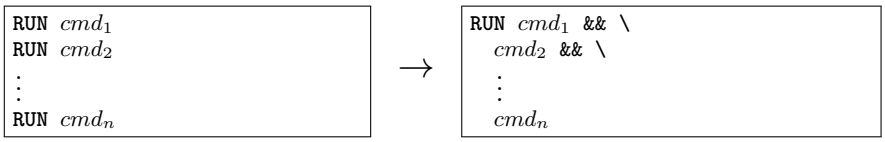


Where $base_1$ and $base_2$ denote image names; $name_1$ denotes the semantic stage alias; $stmt_1, \ldots, stmt_n$ denote Dockerfile statements; and $path_1$ and $path_2$ denote source and destination paths.

Provided $name_1$ is not used as a stage name elsewhere in the Dockerfile, and the index `0` refers to the stage defined by $base_1$; all references to this stage are updated consistently.

3.3.2 T2. Layer Optimization

This transformation merges consecutive `RUN` instructions into a single shell execution context, reducing unnecessary intermediate layers and allowing cleanup commands to take effect within the same layer. It is based on the *Inline Run instructions* refactoring [18].

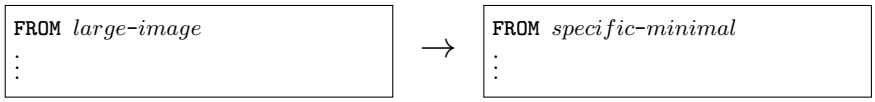


Where cmd_i denotes a shell command; $n \geq 2$.

Provided $cmd_1, \ldots, cmd_n$ preserve their intended behavior when executed sequentially in the same shell context; intermediate layers are not intentionally used for caching, debugging, or isolating failure points.

3.3.3 T3. Update Base Image (Avoid Heavy)

This transformation replaces a large, general-purpose base image with a lighter, purpose-built alternative, reducing unnecessary binaries, libraries, and tools in the final image. It is based on the *Update Base Image* refactoring [18].

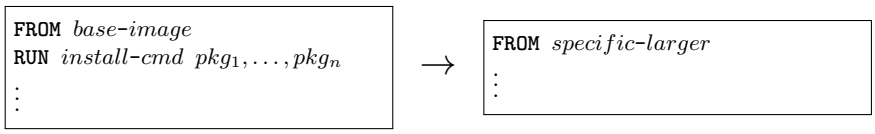


Where *large-image* denotes a general-purpose base image; *specific-minimal* denotes a lighter purpose-built image, such as an Alpine-based or distroless variant.

Provided *specific-minimal* provides the runtime environment and dependencies required by the application; the change does not alter the expected execution behavior of the resulting image.

3.3.4 T4. Update Base Image (Avoid Manual Install)

This transformation replaces a generic base image plus manual dependency installation with a specialized base image that already includes the required dependencies, reducing redundant build steps and improving build reliability when an appropriate image exists. It is based on the *Update Base Image* refactoring [18].

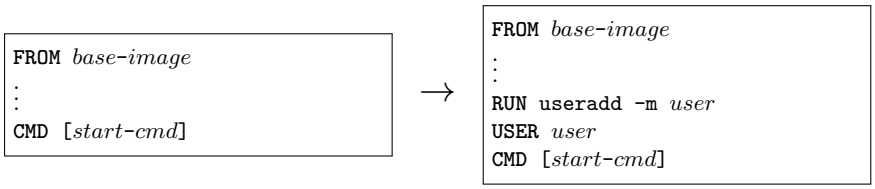


Where *base-image* denotes a generic base image missing required dependencies; *install-cmd* denotes the commands used to install them; pkg_i denotes a required dependency; and *specific-larger* denotes a specialized base image that already includes these dependencies.

Provided *specific-larger* includes all packages $pkg_1, \ldots, pkg_n$ required by the application; the versions and configurations of these packages are compatible with those produced by *install-cmd*; and removing *install-cmd* does not remove dependencies required by later instructions, such as generated configuration files, environment changes, or users.

3.3.5 T5. Avoid Root

This transformation creates a dedicated non-root user and switches the container execution context before the runtime command, reducing the privileges available to the application. It is based on the *Avoid root* practice discussed by Rosa et al. [27].

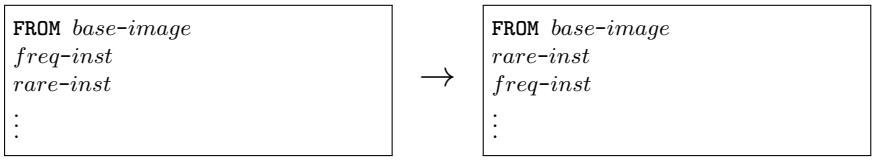


Where *base-image* denotes the base image used by the Dockerfile; *user* denotes the non-root user to be created; and *start-cmd* denotes the command executed when the container starts.

Provided *user* does not already exist in *base-image*; *start-cmd* does not require root privileges; and all files, directories, and ports required at runtime are accessible to *user*.

3.3.6 T6. Sort Instructions

This transformation reorders Dockerfile instructions from less frequently changing to more frequently changing ones, improving layer cache reuse and reducing unnecessary rebuilds of stable layers. It is based on the *Sort Instructions* refactoring [18].

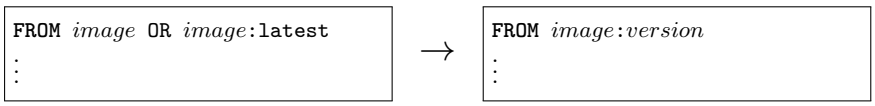


Where *base-image* denotes the image of the stage; *rare-inst* denotes an instruction whose inputs change rarely; and *freq-inst* denotes an instruction whose inputs change frequently.

Provided *rare-inst* does not depend on artifacts produced by *freq-inst*; their relative order does not affect the resulting image.

3.3.7 T7. Explicit Version Pinning

This transformation replaces an implicit or explicit use of `latest` with a specific base image version or digest, reducing build non-determinism and improving reproducibility. It is based on the *Update Base Image TAG* refactoring [18] and addresses the *Explicit Version Pinning* practice discussed by Rosa et al. [27].

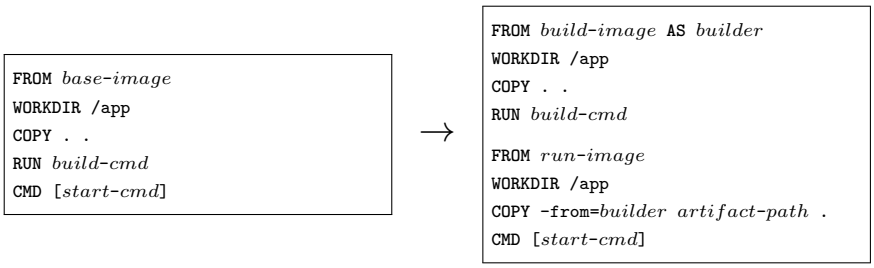


Where *image* denotes a base image; *version* denotes a specific version tag or digest.

Provided *version* identifies the base image version or digest, is available in the target registry, and provides an environment compatible with the one expected by the application.

3.3.8 T8. Extract Stage

This transformation splits a single-stage Dockerfile into a builder stage and a runtime stage, keeping build-time dependencies, source code, and compilation tools out of the final image. It is based on the *Extract Stage* refactoring [18].

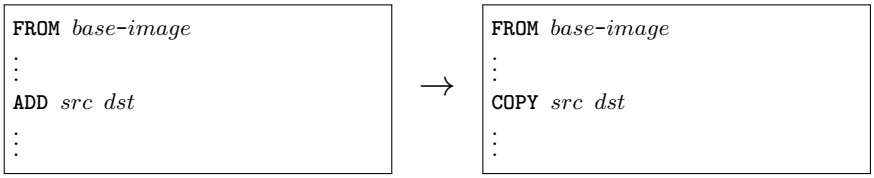


Where *base-image* denotes the original base image; *build-image* denotes the image used to build the application; *builder* denotes the build-stage alias; *build-cmd* denotes the build command; *run-image* denotes the runtime image; *artifact-path* denotes the path of the generated artifacts; and *start-cmd* denotes the command executed at container startup.

Provided *build-image* provides the tools required by *build-cmd*; *build-cmd* produces the application artifacts at *artifact-path*; and *run-image* provides the runtime dependencies required by *start-cmd*; and the final image preserves the runtime-relevant artifacts and configuration required to maintain the original final image behavior.

3.3.9 T9. Replace `ADD` with `COPY`

This transformation replaces `ADD` with the more explicit `COPY` instruction when only local file or directory copying is required, reducing ambiguity and avoiding unintended side effects such as automatic archive extraction. It is based on the *Replace ADD Instruction with COPY Instruction* refactoring [18].

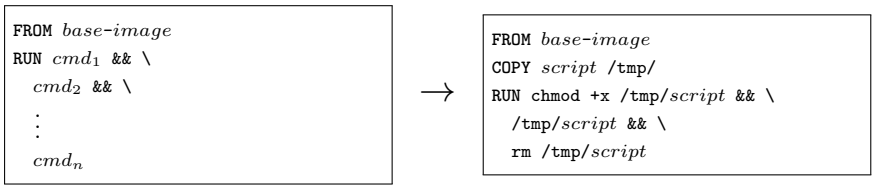


Where *base-image* denotes the base image used by the Dockerfile; *src* denotes the source path in the build context; and *dst* denotes the destination path inside the image.

Provided *src* refers to a local file or directory, not a remote URL; *src* is not an archive whose automatic extraction is required.

3.3.10 T10. Extract `RUN` Instructions

This transformation moves a long sequence of shell commands from an inline `RUN` instruction to an external script, aiming at improving modularity and maintainability of the build logic. It is based on the *Extract Run Instructions* refactoring [18].



Where *base-image* denotes the base image; cmd_i denotes a shell command; $n \geq 2$; and *script* denotes the external shell script containing the extracted commands.

Provided The commands $cmd_1, \ldots, cmd_n$ preserve their intended behavior when executed from *script*; the shell environment used by *script* is compatible with the original inline `RUN` instruction.

4 Empirical Study

After defining the catalog of formalized Dockerfile transformations, we conducted an empirical study to evaluate how practitioners perceive the resulting Dockerfiles. The *goal* of the study was to assess whether the transformations formalized in Section 3 are perceived as useful, understandable, and practical, and whether they are likely to be adopted by developers with Docker experience. The *purpose* is to understand how such transformations improve Dockerfile quality and to identify potential barriers to their adoption in practice. The *perspective* is of researchers and practitioners interested in improving container configuration quality and evaluating the applicability of structured refactoring rules in real-world development contexts. We defined the following two Research Questions (**RQs**) to guide our analysis:

RQ1 To what extent do practitioners prefer Dockerfiles produced by the formalized transformations over the original smelly implementations?
RQ2 Why do developers prefer or reject the transformed Dockerfiles?

RQ1 aims to quantify the degree to which practitioners prefer the transformed Dockerfiles. Since participants were not informed which version corresponded to the transformed one, this question allows us to objectively assess whether the resulting code is naturally perceived as superior to the original implementation.

RQ2 aims to qualify the choices observed in **RQ1** to identify the engineering criteria and trade-offs considered by practitioners when choosing between two Dockerfile implementations, such as readability, build efficiency, maintainability, reproducibility, and security.

To this end, we conducted a mixed-method study combining a survey and semi-structured interviews. We followed empirical software engineering reporting guidelines [13] and the *ACM/SIGSOFT Empirical Standards*[1].

4.1 Research Method

We designed the study as a mixed-method evaluation of the catalog presented in Section 3. For each transformation, we instantiated the formal rule into a pair of Dockerfile snippets: one representing the original smelly implementation and another representing the transformed version. The order of the alternatives was randomized, and participants were not informed

[1] Available at `https://github.com/acmsigsoft/EmpiricalStandards`.

which Dockerfile corresponded to the original or transformed version, to avoid bias. Figure 2 illustrates how the comparison was presented.

A

```
FROM ubuntu:24.04
RUN apt-get update
RUN apt-get install -y python3
RUN apt-get install -y nodejs
RUN apt-get clean
```

B

```
FROM ubuntu:24.04
RUN apt-get update && \
   apt-get install -y python3 nodejs && \
   apt-get clean
```

Figure 2 Example of how Dockerfile versions with and without a smell were presented to participants.

As a mixed-method study, we first conducted a survey to gain a broader quantitative view of practitioners' preferences across the ten transformations. Then, we conducted semi-structured interviews to provide qualitative explanations of the survey results. This design allowed us to investigate both *whether* practitioners preferred the transformed Dockerfiles and *why* they accepted or rejected each transformation.

4.1.1 Survey

As the first empirical method, we conducted a survey with 39 participants, including both practitioners and academics with prior experience writing Dockerfiles.

The survey was designed to collect participants' preferences regarding the Dockerfile versions and optional qualitative comments explaining their choices. The overall study design followed the methodological guidelines proposed by Kitchenham and Pfleeger [15].

The survey target audience consisted of individuals with prior experience creating or maintaining Dockerfiles, whether in professional or academic contexts. We intentionally included participants with diverse levels of experience and expertise to capture perceptions from a heterogeneous Docker user population.

Survey Design

The questionnaire consisted of 24 questions organized into two main categories, as shown in Table 2. The first category collected demographic information about the respondents (SQ1–SQ4). The second category captured participants' perceptions of Dockerfile alternatives through repeated pairwise comparisons, in which participants answered a preference question (SQ5) and could optionally provide an open-ended justification (SQ6). The questionnaire concluded with an acknowledgment section thanking participants for their contribution.

Pilot Test

Before distributing the survey, we piloted the questionnaire with four participants to assess its clarity and estimate completion time. The pilot included two novice Docker users, with 1 and 2 years of experience, and two more experienced users, with 4 and 5 years of experience. Time was 23 and 28 minutes for novice participants and 22 and 24 minutes for experienced participants, resulting in an estimated average of 25 minutes. We then refined the Dockerfiles and instructions to improve clarity.

Table 2 Survey Questionnaire

Category	ID	Question	Answer Format
Demographics	SQ1	How many years of experience do you have with Docker?	Up to 1 – 5 or more
	SQ2	How do you rate your proficiency with Docker?	(Novice) 1 – 5 (Expert)
	SQ3	What is your primary field of experience?	Academic, Professional, Other
	SQ4	What is your level of formal education?	High School – PhD
Perception	SQ5	Which Dockerfile do you prefer?	Strongly A – Strongly B
	SQ6	Any comments?	Open Text Answer

Survey Recruitment

We recruited participants through public invitations posted in online communities related to Docker and software development, including the Docker Official Forum, Discord servers, and Reddit, as well as through internal academic channels. The invitation described the study goal, expected duration, and eligibility criterion: prior experience with Dockerfiles. This strategy is consistent with prior studies on developers' perceptions of refactorings [33].

The recruitment followed a convenience and self-selection sampling strategy. We obtained 39 valid responses, which is comparable to related work with Dockerfile practitioners, such as the study by Rosa et al. [27], which involved 37 participants. Which is adequate for capturing practitioners' perceptions and aligns with the scale of similar studies.

4.1.2 Semi-structured Interview

As the second empirical method, we conducted semi-structured interviews with 20 participants to complement the survey findings with deeper qualitative insights. While the survey provided a broader view of acceptance levels, the interviews were designed to investigate the reasoning behind participants' preferences and to uncover trade-offs or concerns that were not easily captured in closed-ended responses.

Interview Design

The interviews followed a semi-structured protocol [30, 29, 22] divided into three categories. First, we collected the same demographic information used in the survey (IQ1–IQ4) to ensure consistency across methods. Then, participants were presented with the same Dockerfile comparison scenarios and asked to indicate their preferred version (IQ5), explain their reasoning (IQ6), and answer follow-up questions when needed (IQ7). Finally, the interview concluded with a closing section inviting additional remarks and thanking participants. The protocol structure is summarized in Table 3.

Pilot Test

Before the data collection, we piloted the interview protocol with two participants to assess the clarity of the questions, the adequacy of the Dockerfile comparison material, the expected duration, and the recording procedure. Based on the pilot, we clarified the interview questions

Table 3 Interview Protocol

Category	ID	Question / Topic	Answer Format
Demographics	IQ1–4	Same as SQ1–SQ4 in Table 2	Same as SQ1–SQ4
Perception	IQ5	Which Dockerfile do you prefer? (Choice A vs. Choice B)	Open
	IQ6	Why do you prefer this version?	Open
	IQ7	[Follow-up] Could you elaborate on these aspects?	Open

and standardized how sessions were recorded and annotated. No substantial changes were made to the Dockerfile comparison material.

Interview Recruitment and Procedure

We recruited interview participants through convenience and snowball sampling, which are commonly used in empirical software engineering studies involving practitioners [34]. Invitations were distributed through academic mailing lists and shared with students, researchers, and industry contacts, who were asked to forward them to practitioners with experience creating, maintaining, or reviewing Dockerfiles. To mitigate learning bias, we only invited participants who had not previously answered the survey.

We conducted 20 remote interviews via Google Meet between September and October 2025, a number consistent with qualitative empirical studies in software engineering [2]. Each session lasted approximately 20 minutes and, with participants' consent, was audio- and screen-recorded to support transcription and analysis. The interviews complemented the survey by allowing us to go beyond preference distributions and investigate the reasons behind practitioners' preferences, trade-offs, and rejections of specific Dockerfile transformations. All interview recordings were manually transcribed.

4.2 Data Analysis

Once we had completed data collection, we adopted a mixed-methods analysis using quantitative and qualitative techniques. Quantitative analysis was used to characterize participants and summarize stated preferences, while qualitative analysis was used to understand the reasoning behind those preferences.

Quantitative analysis was applied to demographic data and preference choices from both methods. We computed descriptive statistics to characterize participants by Docker experience, self-assessed proficiency, and primary field of work. For the survey (SQ5), responses were aggregated using a binary agree/disagree scheme. For the interviews (IQ5), we categorized participants' explicit preferences for one of the two alternatives. We adopted this aggregation strategy to combine survey and interview data at the transformation level. The resulting data were subsequently aggregated to identify overall acceptance trends.

As for the qualitative analysis of the open-ended survey comments(SQ6) and interview transcripts (IQ6–IQ7), we conducted a thematic analysis following Braun and Clarke [3], focusing on the reasons practitioners provided when accepting or rejecting transformed Dockerfiles. We then coded participants' justifications from the pairwise comparison tasks, assigning each response to an engineering value or constraint (e.g., "READABILITY," "OP-

TIMIZATION," "SECURITY," or "KNOWLEDGE GAP"). To identify distinct behavioral patterns, the coded data were divided into two subsets based on participants' decisions: rationales for agreement (i.e., selecting the transformed Dockerfile) and rationales for disagreement (i.e., preferring the original version). Finally, we quantified the frequency of each theme within these subsets, enabling the identification of the primary drivers of adoption and the main barriers, as discussed in Section 5.2.

5 Analysis of the Results

We gathered data from a total of **59 participants**: 39 from the survey and 20 from the interviews. Table 4 summarizes the distribution of experience, proficiency, and primary fields of work across both groups.

Table 4 Demographic Distribution of All Participants

Characteristic	**Total (N = 59)**
Experience and Proficiency	*Mean ± SD*
Years of Experience	3.37 ± 1.63
Docker Proficiency (1 - 5)	3.42 ± 1.18
Primary Field of Work	*Count (%)*
Professional	51 (86.4%)
Academic	8 (13.6%)

Notably, the study includes a substantial representation of highly experienced engineers: 44% of participants reported 5 or more years of experience, and 23.7% of respondents self-rated their Docker proficiency as expert.

5.1 RQ1: Preference for Transformed Dockerfiles

To answer **RQ1**, we analyzed quantitative data to assess the extent to which developers prefer code generated by the formalized transformations to the original implementations, as shown in Figure 3.

Overall, the transformed versions were preferred in 76.5% of the cases (excluding neutral responses). Acceptance levels varied significantly by smell type, allowing us to categorize the transformations into three groups.

A first group includes transformations with high acceptance rates (above 80%), namely *T1. Multi-stage Build Naming*, *T2. Layer Optimization*, *T3. Update Base Image (Avoid Heavy)*, *T4. Update Base Image (Avoid Manual Install)*, *T7. Explicit Version Pinning*, and *T9. Replace ADD with COPY*. Acceptance rates in this category ranged from 94.7% to 82.5%. A second group comprises transformations with moderate acceptance rates (between 50% and 80%), including *T5. Avoid Root*, *T6. Sort Instructions*, and *T8. Extract Stage*. These transformations were generally well received but generated more disagreement among practitioners, with support ranging from 76.4% for *T5. Avoid Root* to 58.8% for *T6. Sort Instructions*. Finally, *T10. Extract RUN Instructions* was the only transformation with low acceptance (below 50%), receiving an acceptance rate of 25.0%. This indicates a strong preference for keeping build logic inline rather than extracting it into external scripts.

Across all groups, a clear pattern emerges: transformations that provide immediate and recognizable improvements in aspects such as stability, security, and build efficiency

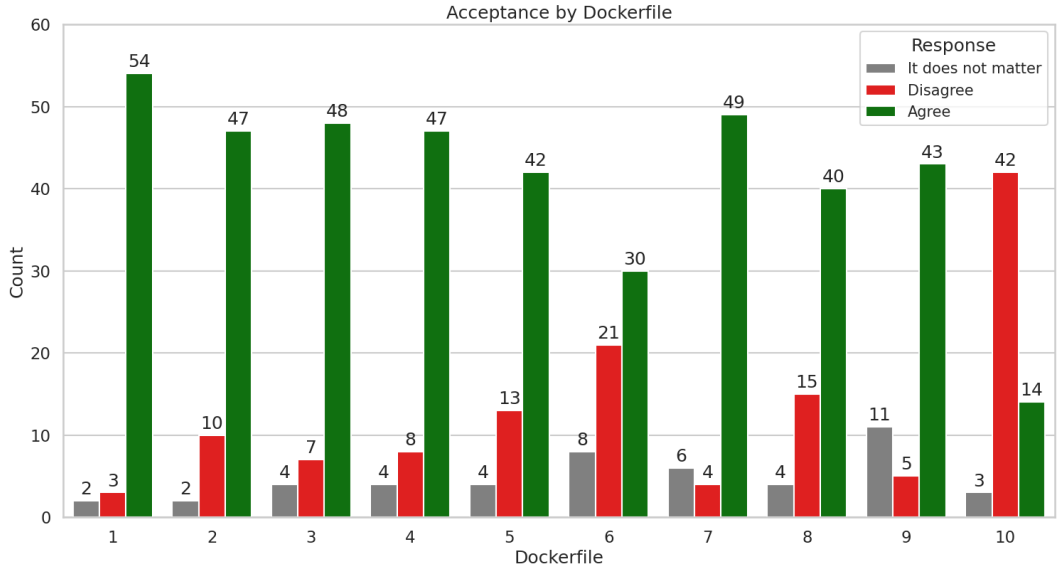


Figure 3 Acceptance of the proposed transformations

tend to be widely accepted, whereas those that introduce structural changes or potential readability trade-offs generate more disagreement. These findings suggest that the acceptance of Dockerfile transformations depends not only on their technical benefits but also on how they impact code readability and developers' workflow.

RQ1 Practitioners generally preferred the transformed Dockerfiles, with an overall acceptance rate of 76.5%. Transformations improving stability, security, and building efficiency achieved the highest agreement, whereas transformations affecting readability or locality of behavior received more mixed reactions. These findings suggest that acceptance of Dockerfile refactorings depends not only on their technical benefits but also on how well they preserve developers' workflows and code inspectability.

5.2 RQ2: Rationale Behind Developer Preferences

The analysis of participants' justifications reveals that preferences are driven by a combination of perceived technical benefits and concerns about code comprehension and workflow. Overall, practitioners prefer transformations when their advantages were concrete and immediately recognizable, while resistance emerged when changes affected readability, debugging, or required deeper knowledge of Docker semantics.

Across reasons for **accepting** transformations, the most frequent motivation was optimization (32%), followed by readability (17%), stability (12%), and security (11%), as shown in Figure 4. Participants consistently highlighted improvements such as reduced image size, fewer layers, faster builds, and increased predictability. For example, in the case of T2 (*Layer Optimization*), respondents explicitly referred to the reduction of layers and resulting smaller images. Similarly, T7 (*Explicit Version Pinning*) was valued for improving reproducibility and preventing unexpected breakages, while T5 (*Avoid Root*) was associated with reducing the attack surface. Overall, participants' justifications aligned with the intended goals of the transformations, suggesting that clearly visible benefits strongly influence acceptance.

At the same time, preferences were shaped by trade-offs, particularly involving readability

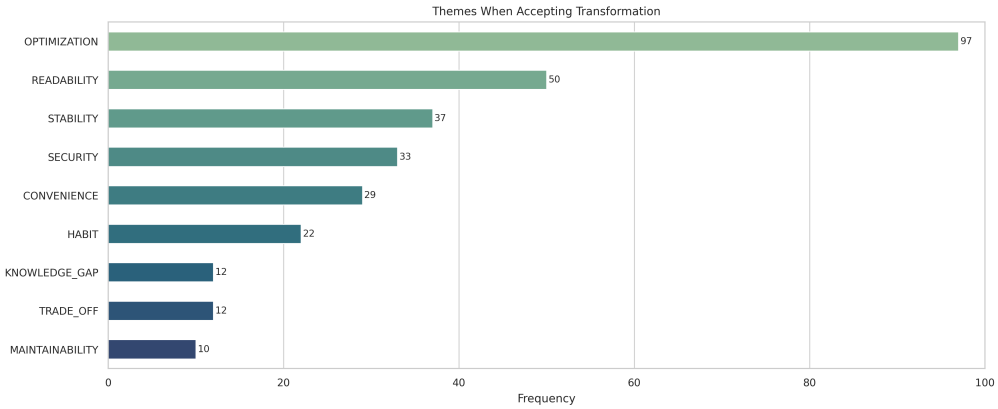


Figure 4 Distribution of themes cited for **accepting** transformations.

and ease of understanding. This factor dominated reasons for **rejecting** transformations, accounting for 42% of disagreement-related themes as reported in Figure 5. Transformations that altered the structure or locality of the Dockerfile were more likely to be questioned. For instance, T10 (*Extract RUN Instructions*) was often perceived as reducing clarity because it moves logic outside the Dockerfile, forcing developers to inspect multiple files. Participants emphasized the importance of keeping build logic in a single place to preserve a linear and inspectable workflow. A similar pattern appears in transformations such as T2 (*Layer Optimization*), where chaining commands improves efficiency but may reduce readability. This indicates that the same transformation can be interpreted differently depending on whether participants prioritize performance or code clarity.

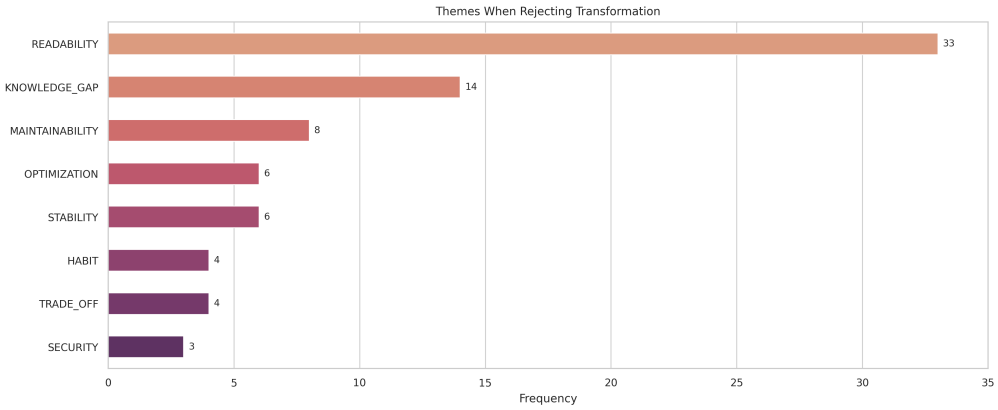


Figure 5 Distribution of themes cited for **rejecting** transformations.

Another recurring factor influencing preferences is the presence of knowledge gaps (18%). Some transformations rely on understanding Docker-specific mechanisms, such as layer caching or instruction semantics. For example, T6 (*Sort Instructions*) was sometimes rejected because participants did not recognize its impact on cache reuse, while T9 (*Replace ADD with COPY*) was sometimes perceived as inconsequential due to limited awareness of their semantic differences. In these cases, resistance does not necessarily reflect disagreement with the transformation itself, but rather uncertainty about its benefits.

Taken together, these results highlight that preferences depend not only on the technical

benefits of a transformation, but also on how it affects the local structure and inspectability of the Dockerfile. This is consistent with prior empirical evidence showing that developers evaluate engineering practices not only in terms of functionality, but also according to their perceived impact on maintainability and development effort [8, 10, 11]. Transformations are more likely to be accepted when they provide clear and immediate benefits without compromising readability or increasing cognitive load. Conversely, when improvements are less visible or introduce structural complexity, practitioners tend to favor solutions that preserve clarity and ease of inspection.

RQ2 Developers primarily accepted transformations for optimization, reproducibility, and security benefits, while rejecting those that reduced readability, debuggability, or the locality of build logic. Knowledge gaps about Docker semantics also influenced several decisions. These results indicate that effective IaC refactoring tools should balance optimization with developer-centric concerns and explicitly communicate the rationale and impact of transformations.

6 Threats to Validity

We discuss potential threats to validity and the strategies adopted to mitigate them.

Construct Validity. In our study, preference is used as a proxy for perceived quality, which may not fully capture real-world effectiveness. Additionally, the binary agree/disagree aggregation may abstract away nuanced opinions. To mitigate these threats, we combined quantitative preferences with qualitative justifications, allowing us to better interpret the underlying reasoning behind participants' choices.

Internal Validity. Potential biases may arise from the study design, such as the presentation of Dockerfiles in isolation and the lack of project context. Participants may also be influenced by personal habits or prior experience. We mitigated these risks by randomizing the order of alternatives, ensuring participants were blind to which version was transformed, and complementing survey data with interviews to validate interpretations.

Conclusion Validity. A key threat concerns the reliability of the observed acceptance trends. The sample size (59 participants) is moderate, and variability in responses across transformations may affect statistical robustness. To mitigate this, we triangulated results across two methods (survey and interviews) and focused on consistent patterns rather than isolated observations. The convergence between quantitative and qualitative findings increases confidence in our conclusions.

External Validity. The generalizability of our results is limited by the participant sample and the study setting. Although most participants are experienced practitioners, the sample may not fully represent the global Docker community. Furthermore, Dockerfiles were evaluated in isolation, whereas real-world decisions are influenced by broader system context. We partially mitigated the threats by leveraging a heterogeneous participant pool and targeting widely used Docker practices, but further studies would strengthen generalizability.

7 Conclusion

We presented a catalog of formalized transformations for removing Dockerfile smells, expressed as explicit rules intended to preserve behavior under stated applicability conditions. We empirically evaluated these transformations through a mixed-method study involving 59 participants, combining quantitative preference data with qualitative insights.

Our findings show that transformations improving determinism, security, and build efficiency are widely accepted, while those affecting readability or locality of behavior are more controversial. This suggests that the adoption of refactoring rules in Infrastructure-as-Code depends not only on technical correctness but also on their impact on developers' workflows and debugging practices.

These results highlight the need for refactoring approaches and tools that balance optimization with developer-centric concerns. Future work includes extending the catalog, evaluating transformations in real-world settings, and improving tool support to make transformations more transparent and actionable.

8 Data Availability

We provide an anonymous replication package available at Zenodo (`https://doi.org/10.5281/zenodo.20260903`). The package includes the study materials, anonymized data, coding schemas and analysis scripts. Raw recordings, comments, and interview transcripts are not released to protect participant confidentiality.

References

1 Manoel Aranda, Naelson Oliveira, Elvys Soares, Márcio Ribeiro, Davi Romão, Ullyanne Patriota, Rohit Gheyi, Emerson Souza, and Ivan Machado. A catalog of transformations to remove smells from natural language tests. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, page 7–16, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3661167.3661225.

2 Sebastian Baltes and Paul Ralph. Sampling in software engineering research: a critical review and guidelines. *Empirical Software Engineering*, 27(4), 2022. doi:10.1007/s10664-021-10072-8.

3 Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3(2):77–101, 2006. doi:10.1191/1478088706qp063oa.

4 Jürgen Cito and Harald C. Gall. Using docker containers to improve reproducibility in software engineering research. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, page 906–907, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2889160.2891057.

5 Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. An empirical analysis of the docker container ecosystem on github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 323–333, 2017. doi:10.1109/MSR.2017.67.

6 José Costa, Rohit Gheyi, Márcio Ribeiro, Sven Apel, Vander Alves, Baldoino Fonseca, Flávio Medeiros, and Alessandro Garcia. Evaluating refactorings for disciplining #ifdef annotations: An eye tracking study with novices. *Empirical Software Engineering*, 26, 07 2021. doi:10.1007/s10664-021-10002-8.

7 Vincenzo De Martino, Gilberto Recupito, Giammaria Giordano, Filomena Ferrucci, Dario Di Nucci, and Fabio Palomba. Into the ml-universe: An improved classification and characterization of machine-learning projects. *Journal of Systems and Software*, 230(C), December 2025. doi:10.1016/j.jss.2025.112471.

8 Sergio Di Meglio, Luigi Libero Lucio Starace, Valeria Pontillo, Ruben Opdebeeck, Coen De Roover, and Sergio Di Martino. Performance testing in open-source web projects: Adoption, maintenance, and a change taxonomy. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 199–210, 2025. doi:10.1109/ICSME64153.2025.00027.

9 Docker Inc. Docker reference – best practices. https://docs.docker.com/build/building/best-practices/, 2026. [Online; accessed 10-May-2026]].

10 Gerardo Festa, Giammaria Giordano, Valeria Pontillo, Massimiliano Di Penta, Damian A. Tamburri, and Fabio Palomba. Pythonic vs refactorable pythonic: On the relationship between pythonic idioms and code quality in machine learning projects. *Information and Software Technology*, page 108009, 2026. URL: https://www.sciencedirect.com/science/article/pii/S0950584925003489, doi:10.1016/j.infsof.2025.108009.

11 Giammaria Giordano, Gerardo Festa, Gemma Catolino, Fabio Palomba, Filomena Ferrucci, and Carmine Gravino. On the adoption and effects of source code reuse on defect proneness and maintenance effort. *Empirical Software Engineering*, 29(1), December 2023. doi:10.1007/s10664-023-10408-6.

12 Giammaria Giordano, Giulia Sellitto, Aurelio Sepe, Fabio Palomba, and Filomena Ferrucci. The yin and yang of software quality: On the relationship between design patterns and code smells. In *2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 227–234, 2023. doi:10.1109/SEAA60479.2023.00043.

13 Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. *Reporting Experiments in Software Engineering*, pages 201–228. Springer, 01 2008. doi:10.1007/978-1-84800-044-5_8.

14 Foutse Khomh, Bram Adams, Tejinder Dhaliwal, and Ying Zou. Understanding the impact of rapid releases on software quality: The case of firefox. *Empirical Software Engineering*, 20, 04 2014. doi:10.1007/s10664-014-9308-x.

15 Barbara A. Kitchenham and Shari Lawrence Pfleeger. Principles of survey research part 2: designing a survey. *ACM SIGSOFT Software Engineering Notes*, 27(1):18–20, January 2002. doi:10.1145/566493.566495.

16 Emna Ksontini, Aycha Abid, Rania Khalsi, and Marouane Kessentini. Drminer: A tool for identifying and analyzing refactorings in dockerfile. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, page 584–594, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3643991.3644921.

17 Emna Ksontini, Thiago Ferreira, Rania Khalsi, and Wael Kessentini. Understanding Docker Refactorings: Expanded Taxonomy, Operational Trade-Offs, and Role-Aware Recommendations . *IEEE Transactions on Software Engineering*, 52(03):786–808, 03 2026. URL: https://doi.ieeecomputersociety.org/10.1109/TSE.2025.3649731, doi:10.1109/TSE.2025.3649731.

18 Emna Ksontini, Marouane Kessentini, Thiago do N. Ferreira, and Foyzul Hassan. Refactorings and technical debt in docker projects: an empirical study. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ASE '21, page 781–791. IEEE Press, 2022. doi:10.1109/ASE51524.2021.9678585.

19 Emna Ksontini, Meriem Mastouri, Rania Khalsi, and Wael Kessentini. Refactoring for Dockerfile Quality: A Dive into Developer Practices and Automation Potential . In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*, pages 788–800, Los Alamitos, CA, USA, 04 2025. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/MSR66628.2025.00116, doi:10.1109/MSR66628.2025.00116.

20 Changyuan Lin, Sarah Nadi, and Hamzeh Khazaei. 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)*, pages 371–381, 2020. doi:10.1109/ICSME46990.2020.00043.

21 Luana Martins, Valeria Pontillo, Heitor Costa, Filomena Ferrucci, Fabio Palomba, and Ivan Machado. Test code refactoring unveiled: where and how does it affect test code quality and effectiveness? *Empirical Software Engineering*, 30(1):27, 2025. doi:10.1007/s10664-024-10577-y.

22 Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP*

2015), volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 495–518, Dagstuhl, Germany, 2015. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2015.495, doi:10.4230/LIPIcs.ECOOP.2015.495.

23 Flávio Medeiros, Gabriel Lima, Guilherme Amaral, Sven Apel, Christian Kästner, Márcio Ribeiro, and Rohit Gheyi. An investigation of misunderstanding code patterns in c open-source software projects. *Empirical Softw. Engg.*, 24(4):1693–1726, 08 2019. doi:10.1007/s10664-018-9666-x.

24 Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. Discipline matters: Refactoring of preprocessor directives in the #ifdef hell. *IEEE Transactions on Software Engineering*, 44(5):453–469, 2018. doi:10.1109/TSE.2017.2688333.

25 Naelson Oliveira, Márcio Ribeiro, Rodrigo Bonifácio, Rohit Gheyi, Igor Wiese, and Baldoino Fonseca. Lint-based warnings in python code: Frequency, awareness and refactoring. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 208–218, 2022. doi:10.1109/SCAM55253.2022.00030.

26 Gilberto Recupito, Giammaria Giordano, Filomena Ferrucci, Dario Di Nucci, and Fabio Palomba. When code smells meet ml: on the lifecycle of ml-specific code smells in ml-enabled systems. *Empirical Software Engineering*, 30(5), July 2025. doi:10.1007/s10664-025-10676-4.

27 Giovanni Rosa, Simone Scalabrino, Gregorio Robles, and Rocco Oliveto. Not all dockerfile smells are the same: An empirical evaluation of hadolint writing practices by experts. In *Proceedings of the 21st International Conference on Mining Software Repositories*, MSR '24, page 231–241, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3643991.3644905.

28 Giovanni Rosa, Federico Zappone, Simone Scalabrino, and Rocco Oliveto. Fixing dockerfile smells: an empirical study. *Empirical Softw. Engg.*, 29(5), 07 2024. doi:10.1007/s10664-024-10471-7.

29 Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, April 2009. doi:10.1007/s10664-008-9102-8.

30 Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, July 1999. doi:10.1109/32.799955.

31 Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does your configuration code smell? In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 189–200, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2901739.2901761.

32 Elvys Soares, Manoel Aranda, Naelson Oliveira, Márcio Ribeiro, Rohit Gheyi, Emerson Souza, Ivan Machado, André Santos, Baldoino Fonseca, and Rodrigo Bonifácio. Manual tests do smell! cataloging and identifying natural language test smells. In *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2023. doi:10.1109/ESEM56168.2023.10304800.

33 Elvys Soares, Márcio Ribeiro, Rohit Gheyi, Guilherme Amaral, and André Santos. Refactoring test smells with junit 5: Why should developers keep up-to-date? *IEEE Trans. Softw. Eng.*, 49(3):1152–1170, 03 2023. doi:10.1109/TSE.2022.3172654.

34 Gabriela Soares, Vanessa Santos, Márcio Ribeiro, Luana Martins, Valeria Pontillo, Manoel Aranda, Rohit Gheyi, Ivan Machado, and Fabio Palomba. On the harmfulness of test smells in manual system testing: A controlled experiment. In *2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, page 185–195. IEEE Press, 2025. doi:10.1109/ESEM64174.2025.00023.

687 **35** Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. Characterizing the occurrence of
688 dockerfile smells in open-source software: An empirical study. *IEEE Access*, 8:34127–34139,
689 2020. doi:10.1109/ACCESS.2020.2973750.
690 **36** Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles, and Jesus M. Gonzalez-
691 Barahona. On the impact of outdated and vulnerable javascript packages in docker images. In
692 *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*
693 *(SANER)*, pages 619–623, 2019. doi:10.1109/SANER.2019.8667984.
694 **37** Ahmed Zerouali, Valeria Pontillo, and Coen De Roover. A comprehensive study of the
695 lifecycle of dormant npm packages. *Empirical Software Engineering*, 31(1), November 2025.
696 doi:10.1007/s10664-025-10753-8.