

# A Catalog of Transformations to Remove Dockerfile Smells

Eduardo Barros 

Federal University of Alagoas, Maceió, Brazil

Valeria Pontillo 

Gran Sasso Science Institute L'Aquila, Italy

Fabio Palomba 

University of Salerno, Fisciano, Italy

Rohit Gheyi 

Federal University of Campina Grande, Brazil

Márcio Ribeiro 

Federal University of Alagoas, Maceió, Brazil

Luana Martins 

University of Salerno, Fisciano, Italy

Baldoino Fonseca 

Federal University of Alagoas, Maceió, 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 source and target structures, and applicability conditions intended to support behavior-preserving application.

**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 a set of ten Dockerfile transformations and conducted a mixed-method empirical study with 59 participants: 39 survey respondents and 20 interviewees from industry and academia with varied levels of Docker experience.

**Results.** Participants generally preferred the resulting Dockerfiles, which achieved an overall mean acceptance rate of 76%. Nine rules were accepted by a majority of participants, with the highest acceptance rate reaching 95%. Participants 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, Transformation, Quality

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.39

**Category** Technical Track Paper

**Supplementary Material** *Collection (Replication Package):* <https://doi.org/10.5281/zenodo.20260903>

**Funding** This work was partially supported by CNPq, grants 312195/2021-4, 404825/2023-0, 443393/2023-0, 301788/2026-0, and 306026/2026-0. Valeria is partially supported by the European HORIZON-KDT-JU research project MATISSE, HORIZON-KDT-JU-2023-2-RIA, Proposal number: 101140216-2, KDT232RIA\_00017.



© Eduardo Barros, Márcio Ribeiro, Luana Martins, Valeria Pontillo, Fabio Palomba, Baldoino Fonseca, Rohit Gheyi, and Ivan Machado;  
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).  
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;  
Article No. 39; pp. 39:1–39:20



Leibniz International Proceedings in Informatics  
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 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 [13, 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 [19], 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, 26, 35, 36]. When these practices are violated, Dockerfiles may contain recurring quality problems, known as *Dockerfile smells* [34]. 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, 34, 26]. Other studies have proposed taxonomies of Docker-specific technical debt categories and refactorings [17, 16]. 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 [17, 16, 25]. 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 evaluate practitioners' perceptions of the resulting Dockerfiles, we conducted a mixed-method empirical study involving 59 participants: 39 survey respondents and 20 interviewees. Our results show that participants preferred the transformed Dockerfiles in 76% of the cases, with acceptance ranging from 95% for the most favored transformation to 25% for the only one rejected by most participants, *Extract RUN Instructions*. Participants strongly favored transformations that enhanced stability, security, and built caching efficiency, but we also observed a tension between technical optimization and human readability: 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 resulting Dockerfiles; and (iii) A qualitative analysis of the reasons why practitioners accept or reject Dockerfile transformations.

## 2 Related Work

The study of Infrastructure-as-Code (IaC) quality has gained significant attention in recent years. Sharma et al. [29] analyzed thousands of Puppet repositories to detect configuration smells, establishing a basis for understanding quality issues in configuration code. In the specific context of containerization, Cito et al. [5] conducted a large-scale study of the Docker ecosystem on GitHub, finding missing version pinning and heavyweight base images among the most common issues. Subsequent work further characterized the prevalence of Dockerfile smells: Wu et al. [34] defined them as instructions violating Dockerfile-specific (DL) or shell-script (SC) best practices, showing in a large-scale GitHub study that such smells are highly prevalent.

Beyond smell detection, researchers have investigated whether Dockerfile smells are relevant to developers and whether they are willing to accept fixes. Rosa et al. [25] evaluated Hadolint smells from the perspective of expert Dockerfile developers, showing that experts prioritize only a subset of the evaluated smells, and proposed an expanded, community-informed catalog. In a complementary study, Rosa et al. [26] analyzed smell-removing commits and submitted pull requests with smell fixes to developers, finding that developers pay more attention to performance-related fixes (e.g., image size, build time) and are generally willing to accept fixes for common smells.

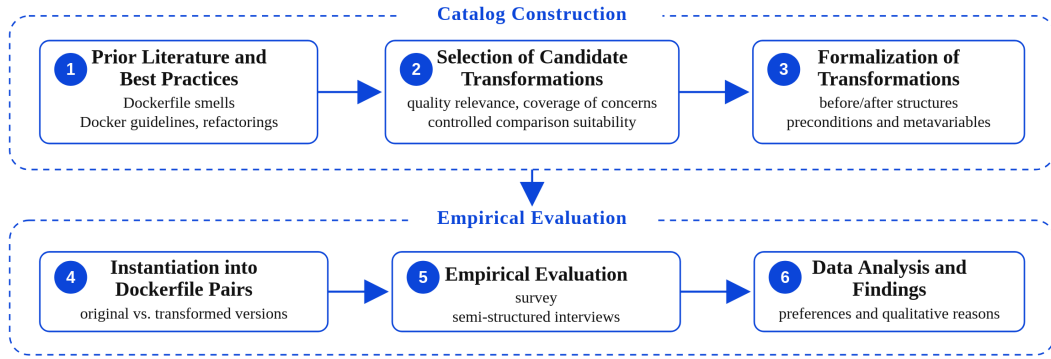
Researchers have also investigated Docker-specific refactoring practices. Ksontini et al. [17] proposed a taxonomy of generic and Docker-specific refactorings based on an analysis of Docker projects, which was later expanded to include additional refactoring mechanisms and technical debt categories [16]. Tool support has also begun to emerge: Ksontini et al. [15] introduced DRMiner, a tool that identifies and analyzes Dockerfile refactoring candidates, and, more recently [18], explored automating Dockerfile refactoring using Large Language Models via In-Context Learning, reporting improvements in image size and build duration.

Although prior studies provide substantial 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 [24].

Our work addresses this gap by formalizing selected fixes as transformation rules and empirically evaluating their perceived quality. Besides making the transformations explicit and actionable, these rules may provide structured demonstrations, constraints, or explanations for LLM-based refactoring approaches such as that proposed by Ksontini et al. [18], and may help assess whether a generated change conforms to an intended transformation. We additionally evaluate practitioners' perceptions of concrete instances derived from these rules. This empirical step is crucial, as even transformations with expected technical benefits 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 concerns the construction of 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 Derivation and Selection of Transformations

We derived the transformation set from prior studies on Dockerfile smells, smell fixes, and Docker-specific refactorings [16, 17, 25, 26, 34]. We first reviewed the problems and refactoring operations reported in these studies and extracted candidates directly applicable to Dockerfiles. When different studies described equivalent or closely related problems, we consolidated them into a single transformation; for example, recommendations concerning missing image tags and the use of the `latest` tag were merged into the *Explicit Version Pinning* transformation. Conversely, when a broader refactoring could address distinct problems through different concrete changes, we kept separate variants: the *Update Base Image* refactoring, for instance, was instantiated as separate transformations targeting heavyweight base images and manually installed dependencies.

We then selected ten transformations according to the following criteria. First, each transformation had to address a Dockerfile smell or refactoring explicitly documented in prior literature. Second, we prioritized transformations targeting smells reported as prevalent in prior empirical studies. For example, *Replace ADD with COPY* (T9) targets the incorrect use of `ADD` for operations that only require local file copying, a smell accounting for 9.04% of the violations reported by Lin et al. [19]; similarly, *Explicit Version Pinning* (T7) addresses untagged base images and the use of the `latest` tag, accounting for 3.30% and 1.90% of the violations reported by the same study. Third, we sought to cover different Dockerfile quality concerns, including readability, maintainability, build efficiency, image size, reproducibility, and security. Fourth, each candidate had to be expressible as a transformation rule with identifiable source and target structures, metavariables, and applicability conditions. Finally, the transformation had to admit a compact before-and-after representation suitable for the controlled pairwise comparisons used in our survey and interviews.

We excluded candidates that were specific to Docker Compose or other orchestration artifacts, required substantial knowledge of the surrounding project, could not be represented through sufficiently explicit applicability conditions, or would require examples too extensive for the empirical instruments. The final set was limited to ten transformations to balance coverage of different quality concerns against the time and cognitive effort required from survey and interview participants.

The resulting catalog is literature-derived and prevalence-informed, but neither exhaustive nor intended as a ranking of the most frequent or severe problems in the Docker ecosystem. Prevalence was one of several selection criteria rather than a uniform requirement, since the source studies differ in datasets, smell definitions, detection tools, and units of analysis. Our goal was a diverse and empirically manageable set of transformations for formalization and controlled evaluation. The findings should therefore be read as characterizing practitioners' perceptions of the ten transformations included in this study, not as evidence that these represent all Dockerfile smells or the ten most important Dockerfile quality problems.

### 3.2 Formalization Procedure

Once the transformations were selected, we formalized them using the template-based representation proposed by Soares et al. [31]. In their approach, refactorings are represented by a left-hand side (LHS), describing the original structure, and a right-hand side (RHS), describing the refactored structure, complemented when necessary by applicability conditions and expected quality improvements.

We adapted this representation to the Dockerfile domain in three ways. First, the structures represented by the rules consist of Dockerfile instructions, build stages, and embedded shell commands rather than traditional source-code or testing-framework constructs. Second, we introduced Docker-specific metavariables for elements such as base images, image tags, stage aliases, commands, paths, package lists, and users. Third, we explicitly represented applicability assumptions through a *Provided* clause, since the applicability of a Dockerfile transformation may depend on both build-time and runtime properties.

For each selected transformation, we followed the same procedure: (i) identify the corresponding smell, fix, or refactoring in the source literature; (ii) determine the minimal Dockerfile structure needed to represent the targeted practice; (iii) express the original and transformed structures as the LHS and RHS, respectively; (iv) introduce metavariables for project-specific elements; and (v) document the applicability assumptions associated with the transformation.

Accordingly, each rule contains four elements: an LHS representing the Dockerfile structure affected by the targeted smell or suboptimal practice; an RHS representing the transformed structure; metavariables abstracting over project-specific elements; and a *Provided* clause specifying the conditions under which the transformation is intended to preserve the relevant build and runtime behavior.

### 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. [16]. 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 [31, 30, 1, 21]. The left-hand side represents the original Dockerfile structure affected by the smell, while the right-hand side represents the transformed structure. Preconditions define

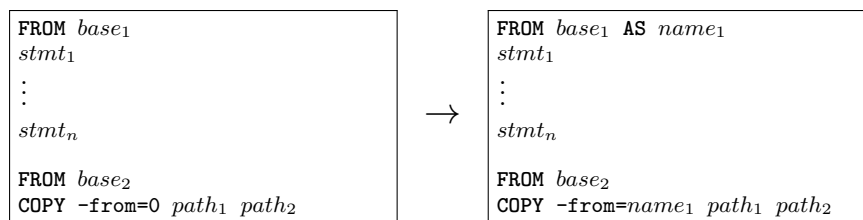
the assumptions required to preserve the intended behavior [22, 6, 23], and metavariables abstract over concrete Dockerfile elements such as image names, commands, paths, stages, and package lists.

■ **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 <code>RUN</code> 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 <code>latest</code> base image tag                            | Reproducibility; stability         |
| T8  | Extract Stage                                   | Single-stage build containing build-time dependencies                    | Image size; modularity; security   |
| T9  | Replace <code>ADD</code> with <code>COPY</code> | Use of <code>ADD</code> when only local copying is needed                | Readability; predictability        |
| T10 | Extract <code>RUN</code> Instructions           | Complex inline shell command sequence                                    | Modularity; maintainability        |

### 3.3.1 T1. Multi-stage Build Naming

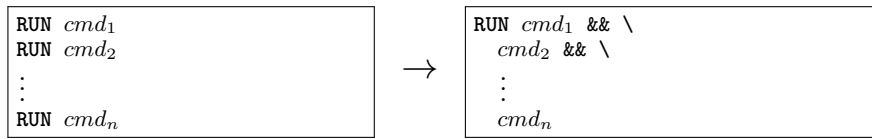
This transformation replaces index-based references to build stages with semantic aliases, reducing coupling between `COPY` instructions and stage order (based on the *Add or rename Image alias* refactoring [17]).



**Where/Provided.**  $name_1$  is the semantic stage alias replacing index 0, which refers to the stage beginning with `FROM base1`;  $name_1$  must not already be used elsewhere as a stage name, and all references to this stage are updated consistently.

### 3.3.2 T2. Layer Optimization

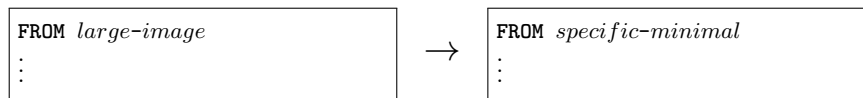
This transformation merges consecutive `RUN` instructions into a single shell execution context, reducing unnecessary intermediate layers and allowing cleanup commands to take effect within the same layer (based on the *Inline Run instructions* refactoring [17]).



**Where/Provided.**  $cmd_1, \dots, cmd_n$  ( $n \geq 2$ ) 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)

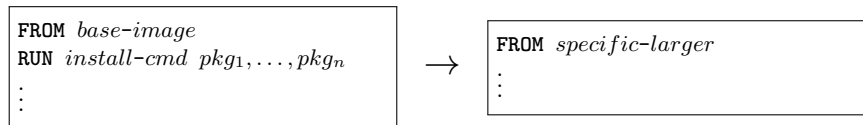
This transformation replaces a large, general-purpose base image with a lighter, purpose-built alternative, reducing unnecessary binaries, libraries, and tools in the final image (based on the *Update Base Image* refactoring [17]).



**Where/Provided.** *specific-minimal* (e.g., an Alpine-based or distroless variant) is a image that provides the runtime environment and dependencies required by the application, without altering the expected execution behavior of the resulting image.

### 3.3.4 T4. Update Base Image (Avoid Manual Install)

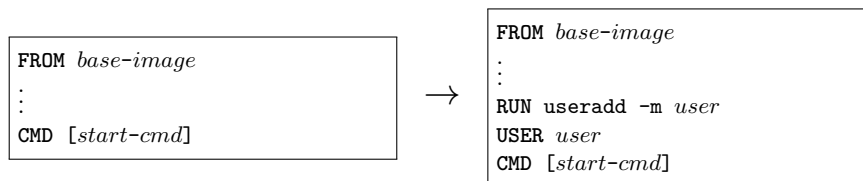
This transformation replaces a generic base image plus manual dependency installation with a specialized base image that already includes the required dependencies, reducing redundant build steps and improving build reliability (based on the *Update Base Image* refactoring [17]).



**Where/Provided.** *specific-larger* is a image that includes all packages  $pkg_1, \dots, pkg_n$  installed by *install-cmd* in *base-image*, with compatible versions and configurations; 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

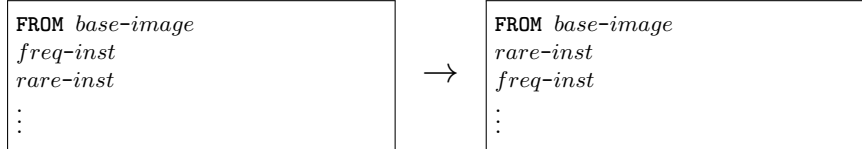
This transformation creates a dedicated non-root user and switches the container execution context before the runtime command, reducing the privileges available to the application (based on the *Avoid root* practice discussed by Rosa et al. [25]).



**Where/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

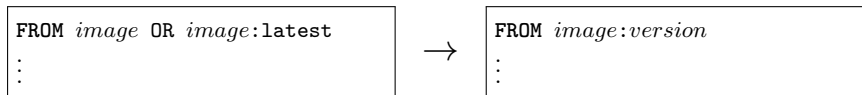
This transformation reorders Dockerfile instructions from less frequently changing to more frequently changing ones, improving layer cache reuse and reducing unnecessary rebuilds of stable layers (based on the *Sort Instructions* refactoring [17]).



**Where/Provided.** *rare-inst* and *freq-inst* denote instructions whose inputs change rarely and frequently, respectively; *rare-inst* does not depend on artifacts produced by *freq-inst*, so their relative order does not affect the resulting image.

### 3.3.7 T7. Explicit Version Pinning

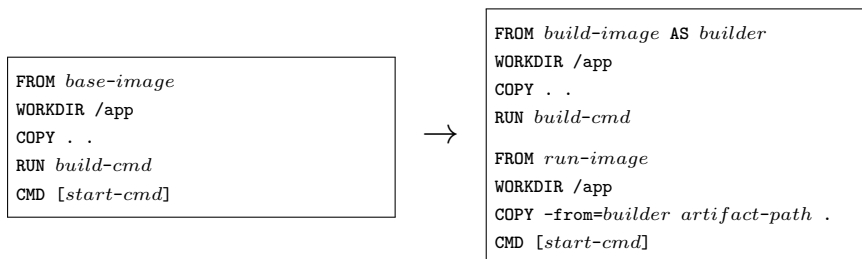
This transformation replaces an implicit or explicit use of `latest` with a specific base image version or digest, reducing build non-determinism and improving reproducibility (based on the *Update Base Image TAG* refactoring [17] and the *Explicit Version Pinning* practice discussed by Rosa et al. [25]).



**Where/Provided.** *version* identifies a specific tag or digest of *image*, 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 (based on the *Extract Stage* refactoring [17]).



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

### 3.3.9 T9. Replace ADD with COPY

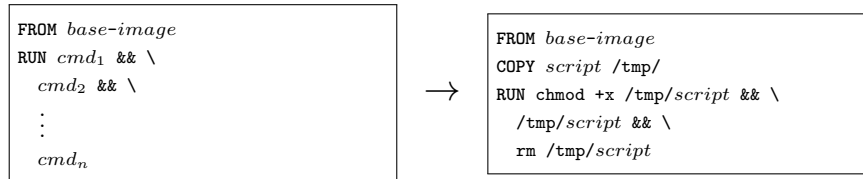
This transformation replaces **ADD** with the more explicit **COPY** instruction when only local file or directory copying is required, reducing ambiguity and avoiding unintended side effects such as automatic archive extraction (based on the *Replace ADD Instruction with COPY Instruction* refactoring [17]).



**Where/Provided.** *src* is a local file or directory in the build context, not a remote URL, and not an archive whose automatic extraction is required; *dst* denotes the destination path inside the image.

### 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 (based on the *Extract Run Instructions* refactoring [17]).



**Where/Provided.**  $cmd_1, \dots, cmd_n$  ( $n \geq 2$ ) preserve their intended behavior when executed from the external script *script*, whose shell environment 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 practitioners' preferences between original and transformed Dockerfile and identify the reasons underlying those preferences. The *purpose* is to understand how such transformations improve Dockerfile quality and to identify potential barriers to their adoption in practice. The *perspective* is that of researchers and practitioners interested in Dockerfile quality and structured refactoring support. We defined the following two Research Questions (**RQs**) to guide our analysis:

**RQ<sub>1</sub>** To what extent do practitioners prefer the transformed Dockerfile over the original?

**RQ<sub>2</sub>** Why do practitioners prefer or reject the transformed Dockerfiles?

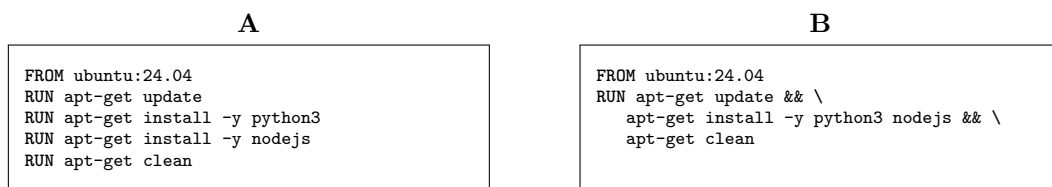
**RQ<sub>1</sub>** quantifies the degree to which practitioners prefer the transformed Dockerfile over the original, using the ten concrete before-and-after instances employed as study stimuli. Since participants were not informed which version corresponded to the transformed one, this design allows us to objectively assess whether the resulting code is naturally perceived as superior to the original implementation. Because each formal rule was represented by a single concrete instance, the results characterize these evaluated instances rather than all possible applications of the corresponding transformation.

**RQ<sub>2</sub>** builds on the preferences observed in **RQ<sub>1</sub>** to qualify the engineering criteria and trade-offs practitioners consider 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 [12] and the *ACM/SIGSOFT Empirical Standards*<sup>1</sup>.

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



■ **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 [14].

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

<sup>1</sup> Available at <https://github.com/acmsigsoft/EmpiricalStandards>.

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.

■ **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 prefer A; Prefer A; Neutral; Prefer B; Strongly prefer B |
|              | SQ6 | Any comments?                                         | Open Text Answer                                                  |

### Pilot Test

We piloted the questionnaire with four participants to assess its clarity and estimate completion time. The pilot involved four participants with varying Docker experience (1–5 years); completion times ranged from 22 to 28 minutes, averaging approximately 25 minutes. We then refined the Dockerfiles and instructions to improve clarity.

### 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 [31].

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. [25], which had 37 participants.

#### 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 [28, 27, 20] 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.

■ **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?<br>(Choice A vs. Choice B) | Open            |
|              | IQ6   | Why do you prefer this version?                            | Open            |
|              | IQ7   | [Follow-up] Could you elaborate on these aspects?          | Open            |

### 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 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 [32]. 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.

For the qualitative analysis, we analyzed the open-ended survey comments (SQ6) and interview responses (IQ6–IQ7) using thematic analysis following Braun and Clarke [3]. The unit of analysis was a participant's justification for one Dockerfile comparison, and a response could receive multiple codes when it addressed multiple engineering concerns.

We then coded participants' justifications from the pairwise comparison tasks, assigning each response to an engineering value or constraint (e.g., "READABILITY," "OPTIMIZATION," "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 us to identify 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. Participants had varying levels of Docker experience: 44% reported five or more years of experience, with 23.7% recognized as Docker experts.

■ **Table 4** Demographic Distribution of All Participants.

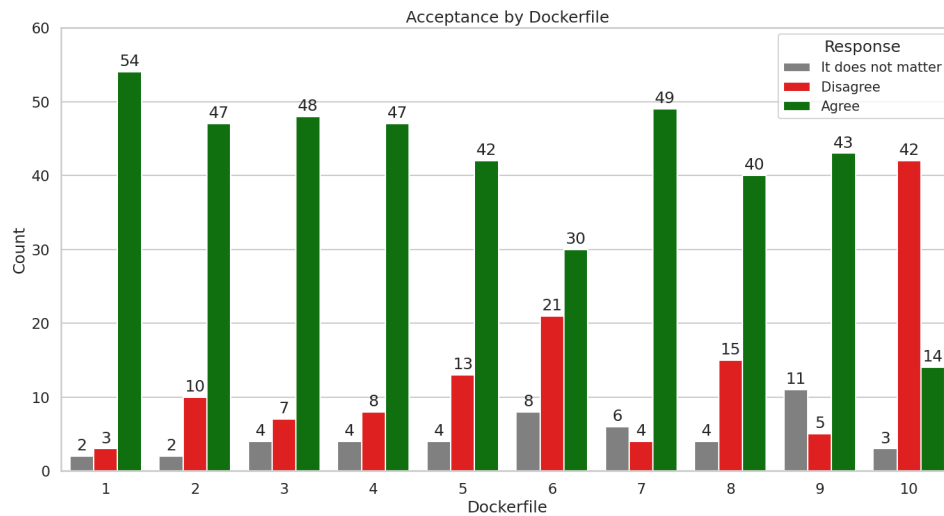
| Characteristic                    | Total (N = 59)   |
|-----------------------------------|------------------|
| <i>Experience and Proficiency</i> | <i>Mean ± SD</i> |
| Years of Experience               | 3.37 ± 1.63      |
| Docker Proficiency (1 - 5)        | 3.42 ± 1.18      |
| <i>Primary Field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

### 5.1 RQ1: Preference for Transformed Dockerfiles

To answer **RQ<sub>1</sub>**, 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, 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



■ **Figure 3** Acceptance of the evaluated transformations.

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. It is important to note that this result should not be generalized to all applications of *Extract RUN Instructions*, as preferences may depend on factors such as the size and complexity of the extracted logic, script reuse, naming, and the surrounding project context.

Across all groups, a clear pattern emerges: transformations that provide immediate and recognizable improvements in stability, security, and build efficiency tend to be widely accepted, whereas those that introduce structural changes or potential readability trade-offs tend to generate more disagreement. These findings suggest that the acceptance of Dockerfile transformations depends not only on their technical benefits but also on their impact on code readability and developers' workflows.

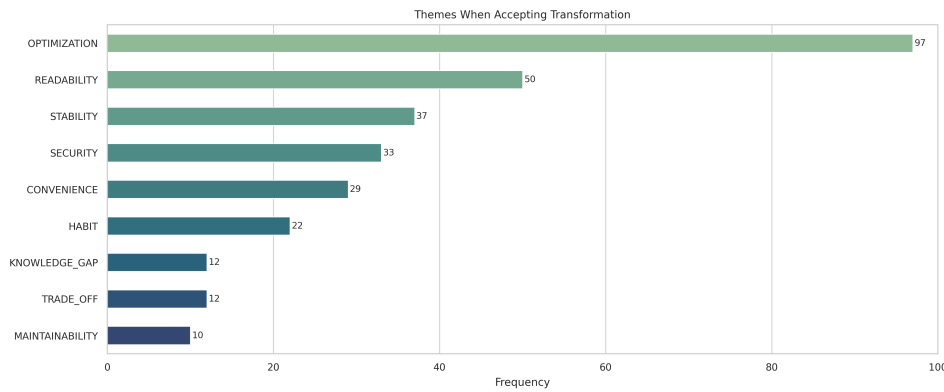
**RQ1.** Practitioners generally preferred the transformed Dockerfiles, with an overall acceptance rate of 76.5%. Transformations improving stability, security, and building efficiency achieved the highest agreement, whereas transformations affecting readability or locality of behavior received more 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 are 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.



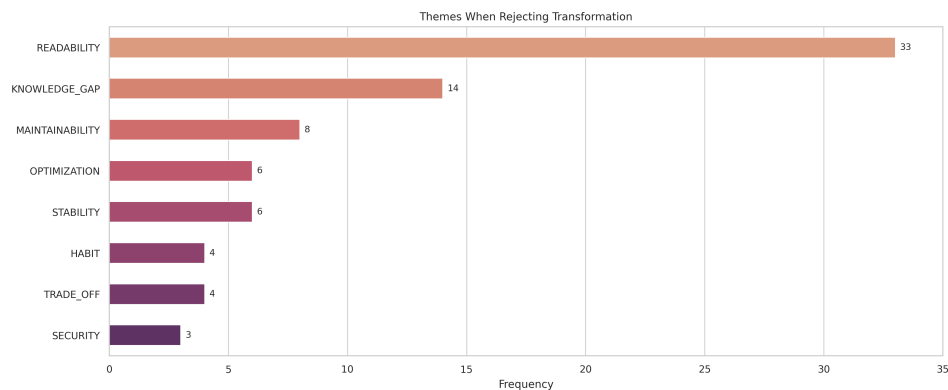
■ **Figure 4** Distribution of themes cited for **accepting** transformations.

At the same time, participants' preferences revealed important trade-offs. Readability and ease of understanding were the dominant reasons for **rejecting** transformed versions, accounting for 42% of the disagreement-related themes shown in Figure 5. Transformations that relied on less visible Docker-specific mechanisms, introduced additional complexity, or changed the Dockerfile's structure tended to receive more mixed evaluations. The following cases illustrate these three distinct sources of disagreement respectively: knowledge of caching semantics (T6), structural complexity (T8), and locality of build logic (T10).

**T6 (Sort Instructions)** received a comparatively mixed response. Qualitative comments show that most participants who preferred the original version did not identify the cache-reuse benefit associated with instruction ordering, whereas those who recognized this mechanism tended to favor the transformed version. This suggests that the perceived value of some transformations depends on whether their effect is visible in the code itself or requires knowledge of Docker-specific mechanisms.

**T8 (Extract Stage)** illustrates a different trade-off. Participants favoring the transformed multi-stage instance emphasized reduced image size and improved security, while those favoring the original version highlighted increased complexity and lower readability. This suggests that multi-stage extraction may be perceived differently depending on whether the reduction in runtime dependencies compensates for the additional structural complexity introduced by multiple stages.

**T10 (Extract RUN Instructions)** produced the strongest disagreement. Participants' comments indicate that moving shell commands to another file introduced indirection and required developers to navigate outside the Dockerfile to understand the build. The result appears to reflect a trade-off between locality and modularity rather than a general rejection of extracting build logic. Since our study used a single concrete extraction, the perceived benefit likely depends on the length, complexity, reuse, and testability of the extracted commands; T10 is thus better interpreted as a context-dependent transformation than as an unconditional fix.



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

Knowledge gaps were the second most frequent factor associated with rejecting transformations (18%). Besides T6 (*Sort Instructions*), T9 (*Replace ADD with COPY*) was sometimes perceived as inconsequential due to limited awareness of the semantic differences between the instructions. Such responses may reflect uncertainty about a transformation’s benefits rather than disagreement with the transformation itself.

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, readability, 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 [33].

**Construct Validity.** Preference is used as a proxy for perceived quality, which may not fully capture real-world effectiveness, and the binary agree/disagree aggregation may abstract away nuanced opinions. We mitigated this by combining quantitative preferences with qualitative justifications. A further threat concerns behavior preservation: the transformation rules include preconditions intended to delimit behavior-preserving applications, but our study evaluates practitioners’ preferences rather than semantic equivalence, and we did not build, execute, or compare the resulting images. Future work should complement this evaluation with build execution and runtime testing.

**Internal Validity.** Potential biases may arise from presenting Dockerfiles in isolation, without project context, and from participants' personal habits or prior experience. We mitigated these risks by randomizing the order of alternatives, keeping participants blind to which version was transformed, and complementing survey data with interviews. Additionally, each rule was represented by a single before-and-after instance, so preferences may have been influenced by the specific commands, images, or formatting used – T10's low acceptance, for example, may partly reflect the particular extraction shown rather than the general concept. Results should thus be generalized to other applications of the same rules with caution; evaluating multiple instances per transformation is left for future work.

**Conclusion Validity.** The sample size (59 participants) is moderate, and response variability across transformations may affect statistical robustness. We mitigated this by triangulating results across survey and interviews and focusing on consistent patterns; the convergence between quantitative and qualitative findings increases confidence in our conclusions.

**External Validity.** Generalizability is limited by the participant sample and study setting. Although the sample included practitioners with different levels of Docker experience, it may not fully represent the global Docker community, and Dockerfiles were evaluated in isolation rather than within broader project context. We partially mitigated this by leveraging a heterogeneous participant pool and targeting widely used Docker practices.

## 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 practitioners' perceptions of concrete instances derived from 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 practitioners' perceptions of Dockerfile refactoring depend not only on expected technical benefits but also on their impact on developers' workflows, readability, 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.

## Data Availability Statement

We provide a 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, Ulyyanne 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, pages 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, pages 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, Balduino Fonseca, Flávio Medeiros, and Alessandro Garcia. Evaluating refactorings for disciplining #ifdef annotations: An eye tracking study with novices. *Empirical Software Engineering*, 26, July 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. 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 Andreas Jedlitschka, Marcus Ciolkowski, and Dietmar Pfahl. *Reporting Experiments in Software Engineering*, pages 201–228. Springer, January 2008. doi:10.1007/978-1-84800-044-5\_8.
- 13 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, April 2014. doi:10.1007/s10664-014-9308-x.
- 14 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.

- 15 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*, pages 584–594, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3643991.3644921.
- 16 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, March 2026. doi:10.1109/TSE.2025.3649731.
- 17 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*, pages 781–791. IEEE Press, 2022. doi:10.1109/ASE51524.2021.9678585.
- 18 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, April 2025. IEEE Computer Society. doi:10.1109/MSR66628.2025.00116.
- 19 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.
- 20 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. doi:10.4230/LIPIcs.ECOOP.2015.495.
- 21 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, August 2019. doi:10.1007/s10664-018-9666-x.
- 22 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.
- 23 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.
- 24 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.
- 25 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*, pages 231–241, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3643991.3644905.
- 26 Giovanni Rosa, Federico Zappone, Simone Scalabrino, and Rocco Oliveto. Fixing dockerfile smells: an empirical study. *Empirical Softw. Engg.*, 29(5), July 2024. doi:10.1007/s10664-024-10471-7.

- 27 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.
- 28 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.
- 29 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*, pages 189–200, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2901739.2901761.
- 30 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.
- 31 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, March 2023. doi:10.1109/TSE.2022.3172654.
- 32 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)*, pages 185–195. IEEE Press, 2025. doi:10.1109/ESEM64174.2025.00023.
- 33 Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al. *Experimentation in software engineering*, volume 236. Springer, 2012.
- 34 Yiwu Wu, Yang Zhang, Tao Wang, and Huaimin Wang. Characterizing the occurrence of dockerfile smells in open-source software: An empirical study. *IEEE Access*, 8:34127–34139, 2020. doi:10.1109/ACCESS.2020.2973750.
- 35 Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles, and Jesus M. Gonzalez-Barahona. On the impact of outdated and vulnerable javascript packages in docker images. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 619–623, 2019. doi:10.1109/SANER.2019.8667984.
- 36 Ahmed Zerouali, Valeria Pontillo, and Coen De Roover. A comprehensive study of the lifecycle of dormant npm packages. *Empirical Software Engineering*, 31(1), November 2025. doi:10.1007/s10664-025-10753-8.