

Article

Repairing Docker Smells with Large Language Models: An Empirical Study

Chenhui Zhang ^{1,2}, Haiyan Wang ² , Junyi Zhu ^{2,3}, Zhaoquan Gu ^{1,2,*}  and Le Wang ^{4,5,*} ¹ School of Computer Science and Technology, Harbin Institute of Technology (Shenzhen), Shenzhen 518055, China; zhangchenhui@stu.hit.edu.cn² Department of New Networks, Pengcheng Laboratory, Shenzhen 518055, China; zhuji01@pcl.ac.cn (J.Z.)³ Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China⁴ Cyberspace Institute of Advanced Technology, Guangzhou University, Guangzhou 510006, China⁵ Huangpu Research School, Guangzhou University, Guangzhou 510006, China

* Correspondence: guzhaoquan@hit.edu.cn (Z.G.); wangle@gzhu.edu.cn (L.W.)

Abstract

Docker simplifies application deployment, yet improperly written Dockerfiles often lead to suboptimal images with security and efficiency issues, termed “Docker smell”. Existing approaches for the identification and repair of Docker smells predominantly rely on expert-defined static rules, which exhibit notable limitations when addressing structurally complex or infrequent smells. This paper proposes a novel Detect–Guide–Repair (DGR) framework, which integrates rule-based smell detection with a context-aware repair mechanism driven by large language models (LLMs), enabling a more flexible and intelligent automated repair process. We systematically evaluated DGR on 417 real Dockerfiles from prominent GitHub projects. Experimental results show that DGR reduces the number of smells to 44.68% of the original while maintaining a build success rate of 89.20%, demonstrating significant improvements in both repair effectiveness and usability. Furthermore, we present three practical enhancement pathways: (1) a hybrid strategy combining rules and DGR to improve repair effectiveness further; (2) an automated error-correction mechanism to restore buildability; and (3) task-specific model fine-tuning to enable efficient deployment of smaller models. Collectively, these approaches provide a promising empirical foundation for automated Docker smell repair.

Keywords: Docker smell; large language model; Dockerfile; empirical study; automated program repair; software maintenance



Academic Editor: Andrea Prati

Received: 11 June 2026

Revised: 30 June 2026

Accepted: 3 July 2026

Published: 7 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

Docker enables developers and organizations to conveniently deploy applications through lightweight containers. These containers are instantiated from Docker images, which are constructed based on instructions specified in Dockerfiles. A Dockerfile defines and automates the image-building process through a series of declarative instructions, rendering its design and implementation critically important [1]. However, the layered and nested semantics involved in Dockerfile authoring often cause engineers without sufficient containerization experience to deviate from established best practices. Such deviations frequently manifest as low-quality Dockerfiles that produce suboptimal container images, leading to inefficiencies in image construction, increased security exposure, and other

operational issues [2–4]. In the literature, these quality issues in Dockerfiles are commonly referred to as Docker smells.

Prior empirical studies have shown that Docker smells are widespread in real-world software repositories and can impose a non-trivial maintenance burden on modern software projects [4–9]. The prevalence of Docker smells has motivated substantial research efforts on automated smell detection. Several static analysis tools and linters have been proposed and widely adopted to identify Dockerfile violations of best practices with relatively high precision [10–14]. While detection techniques have matured, comparatively limited attention has been devoted to the repair of Docker smells.

Existing approaches to Docker smell repair predominantly rely on predefined solutions derived from expert-crafted rules and best-practice guidelines [2,3,15,16]. These approaches are typically implemented as rule-based tools that offer straightforward and interpretable repair. Nevertheless, their effectiveness is generally confined to specific categories of Docker smells and constrained by the expressiveness of expert-defined rules. Empirical observations suggest that rule-based repair strategies often fail to provide comprehensive repair when confronted with complex logic or rare smells. Consequently, many detected smells remain unresolved in practice, limiting the overall practical impact of automated detection tools.

Given the demonstrated success of large language models (LLMs) in program repair [17–19] and the programmatic nature of Dockerfiles, LLM-based approaches present significant potential for automating Docker smell repair. Several recent studies have explored this direction in specific Dockerfile-related tasks, including mitigating build flakiness [20], correcting security misconfigurations [21], and supporting general refactoring [22]. However, these efforts are largely issue-specific and do not provide a systematic understanding of LLM-assisted repair across the diverse spectrum of Docker smells identified by widely used linters such as Hadolint.

This paper presents a systematic empirical study of the Detect–Guide–Repair (DGR) framework. DGR integrates rule-based Docker smell detection with LLM-guided repair: it uses existing linters to precisely identify smells, and then employs LLMs to generate contextually-aware repair. To ground our investigation in realistic development settings, we curate a dataset of 417 Dockerfiles collected from high-star (1000+) repositories on the GitHub platform, reflecting contemporary software engineering practices. Using this dataset, we conduct a rigorous multi-dimensional evaluation of DGR, focusing on its repair effectiveness, practical impact on real-world Dockerfiles, and opportunities for further improvement.

This study is organized around the following research questions:

- **RQ1: How effective is the DGR framework in repairing Docker smells in practice?** This research question examines the practical effectiveness of the proposed framework by evaluating its ability to reduce Docker smells across diverse real-world Dockerfiles.
- **RQ2: How does the repair of Dockerfiles affect the stability of the resulting builds?** This research question investigates the practical impact of Dockerfile repair, specifically evaluating whether the repaired Dockerfiles maintain successful buildability.
- **RQ3: How can the DGR framework be further optimized or extended?** This question explores potential avenues for enhancing the DGR framework, with a focus on hybrid repair strategies, automated error-correction mechanisms, and efficiency optimizations.

Contributions. The major contributions of this work are:

- **Integrated Framework:** We propose DGR, a framework that combines rule-based smell detection with LLM-guided automated Docker smell repair. DGR explicitly uses

precise linter diagnostics to guide the repair process, and we systematically evaluate this guided pipeline on real Dockerfiles.

- **Benchmark Infrastructure:** We construct and publicly release a carefully curated Dockerfile dataset encompassing real-world project scenarios, accompanied by a standardized evaluation metric suite. The entire data collection pipeline and the repair evaluation framework provide a reproducible benchmark for evaluating Docker smell detection and repair techniques, promoting both research and practical application.
- **Empirical Evaluation:** Our evaluation demonstrates that DGR consistently outperforms both purely rule-based and purely LLM-based approaches in Docker smell repair effectiveness. Additionally, DGR maintains high build stability, ensuring that repaired Dockerfiles remain executable in real-world scenarios.
- **Enhancement Pathways:** We identify and empirically validate three concrete avenues for enhancing Docker smell repair systems: hybrid strategies combining rule-based and LLM-guided approaches, automated error-correction mechanisms, and efficiency optimizations. These findings provide actionable guidance for developing more robust and scalable repair solutions in the Docker ecosystem.

2. Background and Key Concepts

Docker leverages OS-level virtualization to package applications and their dependencies into lightweight, portable container images. A Docker image is assembled via a Dockerfile, which serves as a declarative build script specifying how the image should be constructed through a sequence of directives. These directives include selecting a base image (FROM), installing dependencies (e.g., RUN, COPY), configuring the runtime environment (e.g., ENV, WORKDIR), and defining runtime behaviors (e.g., CMD, HEALTHCHECK). Most directives produce new read-only filesystem layers that are stacked sequentially to form the final Docker image.

Despite their apparent simplicity, Dockerfiles are prone to quality issues when best practices are not properly followed. Poorly written Dockerfiles may introduce recurring anti-patterns, commonly referred to as *Docker smells*, which correspond to deviations from established Docker best practices. Prior studies have shown that such smells are prevalent in real-world Dockerfiles and are associated with negative consequences, including bloated images, security vulnerabilities, and non-reproducible builds.

The example Dockerfile shown in Figure 1 (left) illustrates four representative smells that are frequently observed in practice. First, Line 1 uses the ubuntu base image without a version tag, potentially introducing dependency conflicts or breaking changes due to upstream updates. Second, Lines 7–8 install dependencies without performing cleanup, permanently retaining redundant files in the image layers. Third, the Dockerfile (lines 12–13) does not specify a non-root user. As a result, the container will run as root by default, violating the principle of least privilege. Finally, Line 15 does not disable the package manager cache, unnecessarily increasing the final image size. The repaired version in Figure 1 (right) eliminates these smells by pinning the base-image version, introducing cleanup commands, creating an unprivileged user, and disabling the package cache.

This example motivates our work: automatically detecting and repairing Docker smells early in the image life cycle can substantially reduce the resource footprint of continuous integration and deployment pipelines while simultaneously improving build reproducibility and strengthening software supply-chain security.

1	FROM ubuntu	FROM ubuntu:22.04
2	<i># No explicit version tag—future</i>	
3	<i>updates may introduce compatibility</i>	
4	<i>issues</i>	
5	WORKDIR /app	WORKDIR /app
6		
7	RUN apt-get update && \	RUN apt-get update && \
8	apt-get install -y python3-pip	apt-get install -y python3-pip && \
9	<i># APT cache not purged—this</i>	rm -rf /var/lib/apt/lists/*
10	<i>increases the final image size</i>	
11		
12	<i># Running as root by default—may</i>	RUN useradd -m appuser
13	<i>increase security risks in production</i>	USER appuser
14		
15	RUN pip3 install flask ==2.3.2	RUN pip3 install --no-cache-dir \
16	<i>#Pip cache not disabled—this adds</i>	flask ==2.3.2
17	<i>unnecessary layers to the image</i>	
18		
19	COPY app.py /app/	COPY app.py /app/
20	EXPOSE 8080	EXPOSE 8080
21	CMD ["python3", "app.py"]	CMD ["python3", "app.py"]

Figure 1. Left: original Dockerfile with smells. Right: repaired Dockerfile, with changes highlighted in red.

3. Related Work

As Docker has become ubiquitous in modern software development, a growing body of research has investigated Dockerfile quality from multiple perspectives. Existing studies can be broadly categorized into three research streams: (1) Docker smell detection and repair methods, (2) empirical analyses of Docker smells in real-world repositories, and (3) Dockerfile optimization techniques.

Docker smell detection and repair methods. A substantial line of work has focused on systematically detecting Docker smells, which are violations of established Docker best practices. Early studies predominantly rely on pattern mining techniques to infer detection rules from large corpora of Dockerfiles [4,7,9]. For example, Binnacle [4] introduces phased parsing to cope with the nested nature of Dockerfiles, enabling the extraction of abstract syntax trees (ASTs) and the identification of frequent subtree patterns that correspond to best practices. DRIVE [7] further improves detection accuracy by transforming ASTs into instruction sequences and applying frequent sequence mining.

In addition to data-driven approaches, several studies have proposed expert-defined detection rules. Rosa et al. [9], for instance, collaborated with experienced practitioners to curate a prioritized catalog of 26 Docker smells based on expert judgment. In parallel, the developer community has produced a range of widely adopted static analysis tools, including Docker-bench-security [10], Dockerfilelint [11], Dockle [12], and Hadolint [14]. Among these, Hadolint has emerged as a de facto standard Dockerfile linter in both industry and academia and is frequently used as a reference baseline in empirical studies [5,8,9]. Notably, the detection rules implemented by these tools are primarily predefined and curated by domain experts.

Compared to detection, automated Docker smell repair has received relatively limited attention. DockerCleaner [3] targets 11 security-related smells and generates corresponding fixes. More recently, Parfum [2] introduces an advanced AST-based parser to detect and

repair 32 smell types and empirically analyzes the impact of 14 smells on image size. While effective within their scopes, these approaches are predominantly rule-based, which constrains their adaptability to smells involving complex contextual dependencies or rare cases. In contrast, our work explores the use of large language models (LLMs) as a complementary mechanism to support Docker smell repair beyond predefined expert rules.

Empirical analyses of Docker smell. A second stream of research has focused on empirically characterizing Docker smells in large-scale, real-world datasets. Lin et al. [8] analyze over 3.3 million Docker images and 378,615 associated GitHub repositories using Hadolint, observing a gradual decrease in smell prevalence over time, which suggests improving adherence to best practices. Eng et al. [5] report similar findings based on an even larger dataset comprising more than 9.4 million unique Dockerfiles. In the security domain, Liu et al. [23] conduct a comprehensive analysis of Docker images hosted on Docker Hub, revealing widespread security misconfigurations.

More directly related to our research, several studies have examined the practical impact of Docker smells. Durieux et al. [2] identify 16,145 Docker smells across 11,313 open-source Dockerfiles and show that these smells increase image size by an average of 48.06 MB per affected image. Collectively, these empirical studies provide strong evidence that Docker smells are widespread and have tangible negative effects on maintainability, performance, and security. However, they primarily focus on detection and impact analysis, leaving the effectiveness and reliability of automated repair techniques—particularly LLM-assisted approaches—largely unexplored. Our work builds upon these empirical insights by systematically evaluating how LLM-based repair affects Docker image build success and repair outcomes in practice.

Dockerfile Optimization Techniques. Complementary to smell-centric studies, several approaches aim to improve Dockerfile quality through optimization and refactoring. In the context of build efficiency, Doctor [24] optimizes instruction ordering to reduce rebuild time while preserving functional correctness. DRMiner provides tool support for detecting and analyzing refactorings applied to Dockerfiles. More recently, LLM-based techniques have shown promise in improving Dockerfiles across specific dimensions. Shabani et al. [20] propose an iterative LLM-driven approach to mitigate Dockerfile flakiness by combining static analysis with feedback loops. Ye et al. [21] leverage LLMs to repair security misconfigurations, while Ksontini et al. [22] demonstrate that LLM-assisted refactoring can significantly reduce Docker image size.

While these approaches advance Dockerfile quality, they are not explicitly designed to address Docker smells as defined by widely adopted linters such as Hadolint. Moreover, they do not provide a systematic empirical assessment of repair effectiveness across a wide range of smell categories. In contrast, our study focuses on Docker smells as a unifying quality abstraction and conducts a comprehensive empirical evaluation of LLM-assisted smell repair.

4. Research Questions

Our empirical study comprises three research questions aimed at evaluating the effectiveness and impact of LLM in addressing Docker smells.

4.1. RQ1: How Effective Is the DGR Framework in Repairing Docker Smells in Practice?

This research question evaluates the effectiveness of our proposed DGR framework against state-of-the-art baselines. We compare DGR with purely rule-based and purely LLM-based approaches in terms of smell repair rate and the build success rate of repaired Dockerfiles to demonstrate its practical superiority.

4.2. RQ2: How Does the Repair of Dockerfiles Affect the Stability of the Resulting Builds?

Building upon the verification of repair effectiveness (RQ1), this research question evaluates the practical impact of repair operations on Dockerfile buildability. We analyze the build success rate of Dockerfiles repaired by DGR and baseline methods to assess whether the repaired Dockerfiles remain buildable and analyze build failures.

4.3. RQ3: How Can the DGR Framework Be Further Optimized or Extended?

This research question investigates potential pathways for enhancing Docker smell repair systems. Based on our empirical analysis, we explore the efficacy of three specific optimization strategies: hybrid rule–LLM integration, automated error-correction mechanisms, and task-specific model fine-tuning, assessing their potential to improve repair accuracy and deployment efficiency.

5. Methodology

5.1. Baseline Methods

To comprehensively assess the effectiveness of our proposed DGR framework, we compare it against several baseline methods.

5.1.1. Traditional Rule-Based Methods

We evaluate two state-of-the-art rule-based benchmark approaches that represent expert-driven Docker smell detection and repair techniques.

Parfum [2] is an advanced rule-based Docker smell detection and repair tool. It analyzes Dockerfiles using Abstract Syntax Trees (ASTs) enriched with command-line structural information and employs precise pattern matching to detect and fix 32 types of Dockerfile smells.

DockerCleaner [3] is a highly efficient rule-based tool that leverages Hadolint’s AST parser to analyze Dockerfiles. It specializes in detecting and repairing 11 types of security-related Dockerfile smells.

5.1.2. LLM-Based Methods

We selected and reproduced the approach from the most closely related recent work by Ksontini et al. [22], hereinafter referred to as **RE-ICL**, as a key LLM-based baseline. Although the primary objective of RE-ICL is Dockerfile refactoring for image size and build-time optimization, it adopts an in-context learning (ICL) paradigm that leverages large language models to automatically improve Dockerfiles. This methodological foundation is directly relevant to our setting, as Docker smell repair similarly requires context-aware reasoning over Dockerfile instructions. Therefore, RE-ICL serves as a representative and competitive LLM baseline for evaluating the effectiveness of our smell-oriented DGR framework.

5.2. Detect–Guide–Repair (DGR) Framework

DGR is a two-stage framework that integrates rule-based Docker smell detection with LLM-based repair generation. In the first stage, DGR applies Hadolint to perform static analysis on Dockerfiles, identifying smell occurrences together with their corresponding rule identifiers and line numbers (e.g., “:12 DL3059—Multiple consecutive RUN instructions. Consider consolidation.”). These structured detection results are then combined with the original Dockerfile and provided as explicit guidance to the LLM in the second stage, enabling targeted and context-aware repair generation.

By decoupling smell detection from repair generation, DGR leverages the complementary strengths of both paradigms: the determinism and precision of rule-based analysis for

reliable smell localization and the ability of LLMs to generate semantically coherent fixes that go beyond predefined repair templates. The exact prompt template used by DGR has been made publicly available in our replication package in Data Availability Statement. In brief, each prompt supplies the LLM with the original Dockerfile and the corresponding Hadolint diagnostics, and instructs the model to return a modified, buildable Dockerfile that preserves original functionality, without introducing new features.

5.3. Enhancement Strategies for DGR

5.3.1. Hybrid Rule–LLM Repair Strategy

We further explore a hybrid integration strategy that selectively combines rule-based and LLM-based repairs to better exploit the strengths of both approaches. Specifically, for Docker smell types whose repairs are well-defined and consistently supported by existing rules, DGR applies rule-based fixes directly. For cases where rule-based repair is infeasible or the rules provide incomplete coverage, the framework defers the task to the LLM. This conditional dispatch mechanism allows DGR to apply deterministic, low-cost repair where possible while leveraging the LLM's flexibility for more complex or underspecified cases.

5.3.2. LLM Error Correction Mechanism

To mitigate build failures introduced during the repair process, we design an automated failure-driven correction mechanism. Whenever a repaired Dockerfile fails to build, DGR triggers an additional correction step in which the LLM is guided to analyze and resolve the failure. Specifically, the LLM is provided with three structured inputs: (1) the repaired but non-buildable Dockerfile, (2) the corresponding build error log generated by the Docker build process, and (3) the original Dockerfile as contextual reference. The prompt explicitly instructs the LLM to identify the root cause of the build failure and generate a corrected Dockerfile that restores buildability while preserving the previously applied smell fixes.

5.3.3. Task-Specific Fine-Tuning for Efficiency Enhancement

We systematically evaluate the DGR methodology across large language models of different scales, including 235B, 32B, 8B, and 0.6B parameter models from Qwen3, to establish performance baselines. To assess whether task-specific adaptation can enhance repair capabilities for smaller models, we perform fine-tuning using a curated dataset. The dataset is constructed from query-response pairs of original and repaired Dockerfiles, with quality optimized by selecting pairs exhibiting maximal smell score reduction. The effectiveness is evaluated using 5-fold cross-validation to ensure a comprehensive assessment while preventing data leakage.

5.4. Research Framework Overview

Figure 2 illustrates the overall methodology of the DGR framework and its mapping to the corresponding research questions. The original Dockerfile is first analyzed by a static analysis tool (e.g., Hadolint) to detect Docker smells. If smells are detected, the Dockerfile together with the diagnostic information is passed to the LLM for guided repair, producing a revised version. Otherwise, the Dockerfile proceeds directly to evaluation. The framework evaluates the repaired Dockerfile using metrics including Smell-Free Rate, Smell Count, and Smell Scores. This evaluation addresses RQ1. Simultaneously, Docker Build is used to monitor build failures, which corresponds to RQ2.

The lower portion of the figure outlines the optimization process, which addresses RQ3. The original Dockerfile first undergoes rule-based repair, and then proceeds to the DGR framework to improve repair performance. If the repaired Dockerfile fails to build, the LLM performs a correction step to produce a buildable version, thereby enhancing stability.

For speed optimization, we construct a training dataset from repair pairs that yield a large reduction in smell score. This dataset is used to fine-tune smaller, more efficient models, accelerating the repair process. Overall, the framework integrates detection, guided repair, evaluation, and optimization, and visually maps to the three research questions.

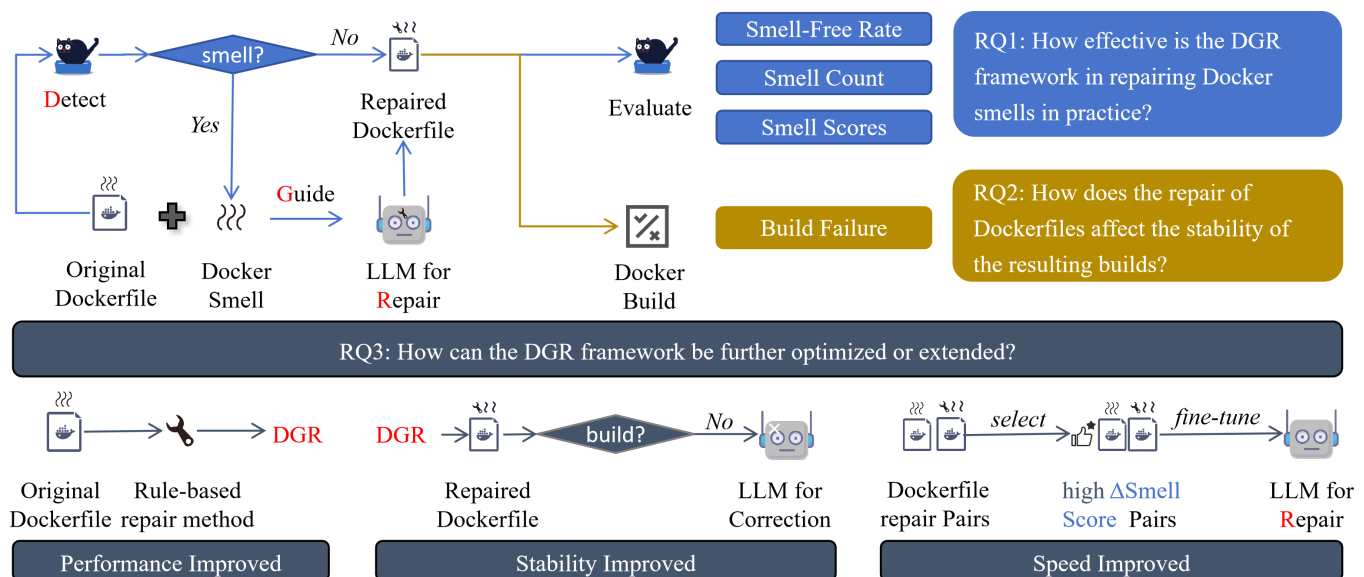


Figure 2. Integrated methodology framework mapping research questions to evaluation approaches.

6. Experimental Setup

This study begins by comprehensively describing our dataset construction pipeline, followed by a detailed specification of the quantitative evaluation metrics. Collectively, these components establish a standardized benchmark framework for comparison of Docker smell repair methods.

6.1. Dataset Construction

We construct a dataset for Docker smell analysis by applying a three-stage screening pipeline to the complete set of GitHub repositories $\mathcal{G}$ [25]. The pipeline uses the following core parameters: star threshold $\theta = 1000$, activity window $\mathcal{T} = [2022.10.01, 2025.10.01]$, per-step time limit $\tau_{\text{step}} = 60$ s, and total build time limit $\tau_{\text{total}} = 600$ s.

Stage 1: Popularity and Activity Filter. We first filter $\mathcal{G}$ for repositories with more than θ stars and recent activity within $\mathcal{T}$, yielding 22,720 repositories ($\mathcal{R}$).

Stage 2: Dockerfile Presence Filter. We then process each repository in $\mathcal{R}$ to verify the presence of a Dockerfile in its root directory. This step identifies 2557 projects ($\mathcal{R}^D$).

Stage 3: Build Efficiency Filter. For each project in $\mathcal{R}^D$, we execute its docker build and measure step durations. A project is retained only if no single step exceeds τ_{step} and the total build time is within τ_{total} .

The complete construction logic is formalized in Algorithm 1. The final dataset $\mathcal{R}^F$ contains 417 high-quality projects that balance representativeness with practical research efficiency.

Algorithm 1 Dataset Construction Pipeline

Require: GitHub repository set $\mathcal{G}$, star threshold $\theta = 1000$, time window $\mathcal{T} = [2022.10.01, 2025.10.01]$, step time limit $\tau_{\text{step}} = 60$ s, total build time limit $\tau_{\text{total}} = 600$ s

Ensure: Filtered repository set $\mathcal{R}^F$

- 1: $\mathcal{R} \leftarrow \emptyset, \mathcal{R}^D \leftarrow \emptyset, \mathcal{R}^F \leftarrow \emptyset$
- 2: **Stage 1: Popularity and Activity Filter**
- 3: $\mathcal{R} \leftarrow \{r_i \in \mathcal{G} \mid \text{stars}(r_i) > \theta \wedge \exists c \in \text{commits}(r_i), c.\text{date} \in \mathcal{T}\}$
- 4: **Stage 2: Dockerfile Presence Filter**
- 5: **for** each $r_i \in \mathcal{R}$ **do**
- 6: $\mathcal{F}_i \leftarrow \{f \in \text{files}(r_i) \mid \text{depth}(f) = 0\}$
- 7: **if** $\exists f \in \mathcal{F}_i$ such that isDockerfile(f) **then**
- 8: $\mathcal{R}^D \leftarrow \mathcal{R}^D \cup \{r_i\}$
- 9: **end if**
- 10: **end for**
- 11: **Stage 3: Build Efficiency Filter**
- 12: **for** each $r_i \in \mathcal{R}^D$ **do**
- 13: $\text{step_durations} \leftarrow \text{docker build}(r_i)$
- 14: **if** $\max(\text{step_durations}) \leq \tau_{\text{step}} \wedge \sum(\text{step_durations}) \leq \tau_{\text{total}}$ **then**
- 15: $\mathcal{R}^F \leftarrow \mathcal{R}^F \cup \{r_i\}$
- 16: **end if**
- 17: **end for**
- 18: **return** $\mathcal{R}^F$

6.2. Evaluation Metrics

We employ a multi-dimensional evaluation framework comprising two main categories of metrics: (1) metrics related to the effectiveness of smell repair, and (2) metrics related to the practical impact after smell repair.

6.2.1. Metrics for Repair Effectiveness

We evaluate the effectiveness of Docker smell repair through three primary metrics: the **Smell-Free Rate**, **Smell Count**, and **Smell Score**. Each metric provides a distinct perspective on repair quality.

Smell-Free Rate quantifies the ultimate goal of smell repair by measuring the percentage of Dockerfiles that become completely free of smells after repair. This metric is calculated as:

$$\text{Smell-Free Rate} = \frac{N_{\text{smell-free}}}{N_{\text{total}}} \times 100\% \quad (1)$$

where $N_{\text{smell-free}}$ is the number of Dockerfiles with zero smells after repair, and N_{total} is the total number of Dockerfiles evaluated.

Smell Count provides a direct measure of the quantitative reduction in smells. We analyze this metric from two perspectives: first, smell severity, which is used to assess the criticality of its potential impact; second, functional impact, which evaluates the Docker capabilities that may be affected.

To account for the varying severity of different smell types, we classify them into three levels based on Hadolint's documentation and manual definitions:

- **Errors:** Critical failures requiring immediate attention due to security vulnerabilities or build failures (e.g., DL3004 prohibiting sudo usage, DL3020 recommending COPY over ADD for security).
- **Warnings:** Non-critical issues affecting maintainability, performance, or adherence to best practices (e.g., DL3007 warning against latest tags, DL3027 recommending apt-get over apt for scripting).

- **Infos:** Optimization opportunities and code quality improvements enhancing efficiency and maintainability (e.g., DL3015 suggesting `-no-install-recommends`, DL3059 recommending RUN command consolidation).

These severity-weighted categories enable the calculation of the **Smell Score**, defined as:

$$\text{Smell Score} = 5 \times N_{\text{error}} + 3 \times N_{\text{warning}} + 2 \times N_{\text{info}} \quad (2)$$

where N_{error} , N_{warning} , and N_{info} represent the counts of error, warning, and info level smells, respectively. This scoring system provides a comprehensive measure of the overall smell severity in a Dockerfile, with higher weights assigned to more critical smell types.

From a functional impact perspective, we classify Docker smells into four key dimensions that correspond to different quality attributes:

- **Security:** Smells that compromise the security of Docker images and containers. This classification follows the definitions in Dockercleaner [3]. Examples include DL3002 (prohibiting root user execution) and DL3026 (enforcing trusted registry sources).
- **Correctness:** Smells that may lead to build or runtime failures and behavioral anomalies. Examples include DL3011 (validating port ranges) and SC1097 (addressing incorrect comparison operators in shell scripts).
- **Maintainability:** Smells affecting metadata validity and code clarity. Examples include DL3048 (enforcing label key syntax) and DL4000 (deprecating obsolete MAINTAINER instructions).
- **Efficiency:** Smells causing image bloat, slow builds, or resource waste. Examples include DL3009 (mandating APT cache cleanup) and SC2002 (eliminating redundant cat operations).

6.2.2. Metrics for Practical Impact

Beyond smell elimination effectiveness, we assess the practical impact of repair methods through two key metrics: **build success rate** and **build failure categorization**.

The **build success rate** measures the ability of repaired Dockerfiles to maintain build integrity and is calculated as

$$\text{Build Success Rate} = \frac{N_{\text{successful builds}}}{N_{\text{total builds}}} \times 100\% \quad (3)$$

where $N_{\text{successful builds}}$ is the number of Dockerfiles that build successfully after repair, and $N_{\text{total builds}}$ is the total number of Dockerfiles evaluated. This metric directly reflects the practical viability of the repair process.

To better understand the types of build failures that may occur after repair, we categorize them into four distinct types based on Docker build stages:

- **Base image stage errors:** Failures occurring during initial image setup, including unavailable base images, incompatible image versions, or registry access issues.
- **Context stage errors:** Issues related to file handling in the Docker build context, such as missing source files, incorrect paths in COPY/ADD instructions, or permission conflicts.
- **Command execution stage errors:** Failures during command execution, including package installation failures, dependency conflicts, compilation errors, or script execution problems.
- **Environment configuration stage errors:** Syntax errors, invalid instruction formats, environment variable misconfigurations, or improper Dockerfile structure.

This systematic categorization enables precise identification of failure patterns, which helps in understanding the limitations and potential side effects of different repair methods.

7. Experimental Configuration

This section details the experimental setup for comparing traditional and LLM-based Dockerfile repair methods.

For the LLM-based approach, we employ the qwen3-235b-a22b-instruct-2507 model [26]. The model is configured with a temperature of 0.3 to balance output consistency with necessary flexibility and a maximum token limit of 4096. All other parameters remain at their default values.

For traditional repair methods, we use the default configurations provided in their respective implementations. This ensures a fair comparison under each method's intended operating conditions, avoiding any configuration bias.

The evaluation protocol uses Hadolint to detect smells in the repaired Dockerfiles. We parse Hadolint's output and map it to the evaluation metrics defined in Section 6.2. This provides a standardized framework for assessing and comparing repair effectiveness across methodologies.

To strengthen the rigor of the comparison, we perform Dockerfile-level statistical significance tests. Since each method is evaluated on the same 417 Dockerfiles, we treat each Dockerfile as a paired observation. For Smell Count and Smell Score, we use the one-sided Wilcoxon signed-rank test to examine whether DGR yields lower values than each baseline. For Smell-Free Rate, which is binary at the Dockerfile level, we use McNemar's exact test. We apply Bonferroni correction for the three main baseline comparisons.

Training uses a maximum sequence length of 4096, per-device batch size: 2, gradient accumulation steps: 4, 3 epochs, learning rate: 2×10^{-5} , 100 warm-up steps, bfloat16 precision, gradient checkpointing, and a single-model checkpoint retention policy. During inference, we use greedy decoding with max_new_tokens=512, num_beams=1.

8. Study Results

This section presents the experimental results addressing our research questions on Docker smell repair.

8.1. RQ1: Effectiveness of Docker Smell Repair

Table 1 compares the effectiveness of different repair methods against the original Dockerfiles using three overarching metrics: Smell Count, Smell Score, and Smell-Free Rate. The results demonstrate that our proposed DGR approach outperforms all baseline methods across all three metrics.

Table 1. Overall repair effectiveness of different methods. Lower (↓) is better for Smell Count and Score; higher (↑) is better for Smell-Free Rate.

Method	Smell Count ↓	Smell Score ↓	Smell-Free Rate ↑
Original	1099 (100.00%)	3105 (100.00%)	19.90%
DockerCleaner	919 (83.62%)	2539 (81.77%)	22.78%
Parfum	740 (67.33%)	2120 (68.28%)	26.37%
RE-ICL	642 (58.42%)	1834 (59.07%)	29.49%
DGR	491 (44.68%)	1452 (46.76%)	35.73%

DGR reduces the Smell Count to 44.68% of the original level, representing an improvement of 10.56 percentage points over the best baseline (RE-ICL). The more pronounced reduction in the weighted **Smell Score** indicates that DGR is particularly effective at fixing higher-severity smells. Furthermore, DGR achieves the highest **Smell-Free Rate**, producing the greatest number of completely smell-free Dockerfiles.

We conducted Dockerfile-level statistical tests to compare DGR against three baselines. Using one-sided Wilcoxon tests for Smell Score and Smell Count and McNemar's exact test for Smell-Free status, we applied Bonferroni correction ($\alpha = 0.0167$). DGR significantly outperformed all baselines across all metrics: versus DockerCleaner ($p = 1.10 \times 10^{-32}$, $p = 2.77 \times 10^{-34}$, $p = 9.00 \times 10^{-13}$ respectively), versus Parfum ($p = 1.29 \times 10^{-22}$, $p = 1.31 \times 10^{-21}$, $p = 8.65 \times 10^{-10}$), and versus RE-ICL ($p = 3.57 \times 10^{-8}$, $p = 2.31 \times 10^{-9}$, $p = 2.55 \times 10^{-3}$). These results demonstrate that the observed improvements are not merely aggregate-level differences, but are consistently supported by paired Dockerfile-level evidence.

Figure 3 provides a detailed breakdown of the **Smell Count** reduction across seven sub-categories. DGR achieves the best performance in six of the seven categories, demonstrating comprehensive repair capability. In the remaining category (Security), it delivers competitive results. It can be observed that both traditional rule-based methods and large language model (LLM)-based methods prioritize fixing Error-type smells, which aligns with practical maintenance priorities. LLM-based methods generally perform well, showing particular strength in addressing smells related to Maintainability. This is likely because such smells are often less critical, with no immediate impact, and are therefore frequently overlooked by developers of rule-based tools. Compared to the RE-ICL method, which is also LLM-based, our DGR approach demonstrates superior performance. This advantage can likely be attributed to the strategic prompts provided to the model regarding the specific smell categories requiring repair, leading to more precise fixes.

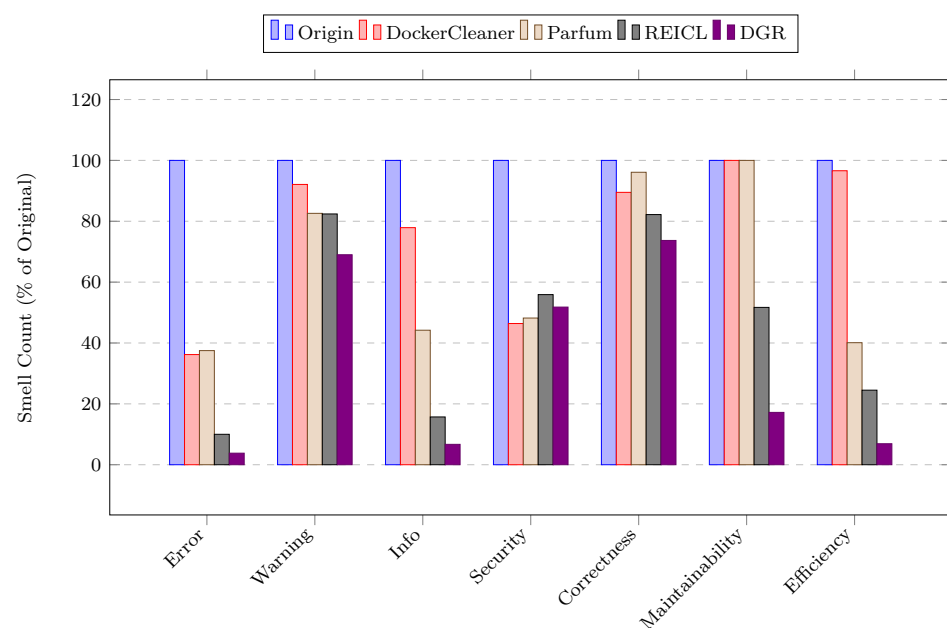


Figure 3. Comparison of repair effectiveness across seven smell categories. Values are normalized to the original count (100%). Lower is better.

The above analysis underscores the importance of repair prioritization in practice, a factor well-captured by the **Smell Score** metric. The pronounced decrease in the Smell Score indicates that the DGR method is particularly effective at identifying and rectifying higher-priority smells, thereby significantly reducing the overall severity.

The high **Smell-Free Rate** of DGR stems from the contextual understanding of LLMs. Unlike rule-based tools that apply targeted fixes, LLMs can comprehend the entire Dockerfile to perform a holistic cleanup. Furthermore, as DGR is specifically tailored for smell repair rather than general restructuring, it achieves a better cleanup rate than RE-ICL.

8.2. RQ2: Build State of the Dockerfile After Docker Smell Repair

Our experimental results show that Docker smell repair—whether using traditional rule-based methods or LLM-based approaches—can frequently undermine build stability.

For example, the Dockercleaner method enforces security hardening by systematically introducing unprivileged users via the instruction sequence: `RUN groupadd -system docker-user && useradd -system -gid docker-user docker-user`. However, this leads to build failures in Alpine-based images, as their minimal BusyBox environment lacks essential user management commands such as `useradd` and `groupadd`. Such errors are often difficult to detect with static analysis tools, since the instructions themselves are syntactically valid.

Similarly, the Parfum method applies a common optimization rule: cleaning the APT cache immediately after installation to reduce image size. It generates instructions like: `RUN apt-get update && apt-get install -y nginx && rm -rf /var/lib/apt/lists/*`. However, when multiple installation steps are required—e.g., a subsequent `RUN apt-get install -y vim`—this premature cache cleanup causes the build to fail. The removal of the cache directory eliminates package index information necessary for later dependency resolution.

In LLM-based repair methods, a common failure pattern arises from the model's tendency to hallucinate. Although LLMs can process the full context of a Dockerfile, they often generate instructions based on incomplete or inaccurate domain knowledge. For instance, they may be unaware of package availability or specific base image versions, leading to invalid recommendations. Additionally, insufficient contextual awareness can result in incorrect instruction merging that disrupts critical execution order. These cases highlight that Docker smell repair strategies must be carefully adapted to the specificities of the base image, the build context, and the operational constraints of the target environment.

Table 2 reports the build success rates and the distribution of error occurrence stages for each method. The table uses the following abbreviations: “Img.” for base image stage errors, “Ctx.” for context stage errors, “Exec.” for command execution stage errors, and “Env.” for environment configuration stage errors.

Table 2. Build success rates and error distribution by stage after repair.

Method	Success Rate	Img.	Ctx.	Exec.	Env.
DockerCleaner	234/417 (56.11%)	4	1	177	1
Parfum	380/417 (91.12%)	5	0	32	0
RE-ICL	226/417 (54.19%)	41	26	121	3
DGR	372/417 (89.20%)	6	3	36	0

The data in Table 2 reveals distinct performance profiles for each method. Parfum and DGR achieve the highest build success rates (91.12% and 89.20%, respectively), with most failures occurring at the command execution stage. This indicates that their targeted strategies, while not flawless, generally preserve build integrity. In contrast, DockerCleaner's lower success rate (56.11%) is dominated by command execution errors, highlighting the risks of applying rigid rules across diverse environments.

RE-ICL, employed as a general-purpose refactoring tool rather than a smell-specific repair method, demonstrates the most distributed error profile and the lowest success rate (54.19%). Its attempts to rewrite the Dockerfile frequently introduce fundamental errors at the base image and context stages, in addition to command execution issues. This suggests that without explicit guidance focused on smell correction, LLM-driven refactoring can inadvertently disrupt the build by making overreaching or incorrect changes.

8.3. RQ3: Empirical Analysis of Enhancement Strategies for DGR

This section empirically evaluates three enhancement strategies for the DGR framework: a hybrid rule–LLM repair pipeline, an LLM-based error correction mechanism, and task-specific fine-tuning for efficiency.

8.3.1. Evaluation of the Hybrid Rule–LLM Repair Strategy

We evaluate a hybrid repair strategy that combines rule-based and LLM-based methods, comparing it against pure rule-based (Parfum) and our proposed DGR LLM-based method. Table 3 presents the results.

Table 3. Repair effectiveness of rule-based (Parfum), LLM-based (DGR), and hybrid (Parfum + DGR) strategies.

Metric	Parfum	DGR	Parfum + DGR
Smell Count	740 (67.33%)	491 (44.68%)	448 (40.80%)
Smell Score	2120 (68.28%)	1452 (46.76%)	1339 (43.12%)
Smell-Free Rate	26.37%	35.73%	37.88%
LLM API Calls	0	334	309

The hybrid strategy achieves the best performance across all metrics. It reduces the smell count to 40.80% of the original, compared to 67.33% for Parfum and 44.68% for DGR. Similarly, it yields the lowest smell score (43.12%) and the highest smell-free rate (37.88%). This indicates that combining rule-based and LLM-based repair can handle a wider variety of smell patterns and produce more complete fixes.

A key advantage is efficiency: while pure DGR requires 334 LLM API calls, the hybrid strategy needs only 309, achieved by using Parfum for rule-coverable smells and invoking DGR only for complex cases. The performance gain stems from the complementary strengths of both approaches: Parfum provides deterministic fixes for pattern-based smells, while DGR handles complex semantic smells requiring contextual understanding.

8.3.2. Evaluation of the LLM-Based Error Correction Mechanism

As identified in RQ2, automated smell repair can break Dockerfile buildability. We therefore repurpose the broken Dockerfiles from RQ2 to evaluate an LLM-based correction strategy. Table 4 summarizes the results.

Table 4. Build success rates before and after LLM-based correction.

Method	After Repair	After Correction	Correction Rate
Parfum	380/417 (91.12%)	387/417 (92.81%)	7/37 (18.91%)
DGR	372/417 (89.20%)	384/417 (92.08%)	12/45 (26.66%)

The corrector successfully fixed 18.91% of Parfum-induced failures and 26.66% of DGR-induced failures. Consequently, the final build success rates increased to 92.81% for Parfum and 92.08% for DGR. Notably, the pure LLM pipeline (DGR repair + LLM correction) achieves a success rate (92.08%) that surpasses the rule-based Parfum method after repair alone (91.12%). This demonstrates the feasibility and competitiveness of an LLM-driven repair system, where the correction step acts as a safety net for build breaks.

8.3.3. Evaluation of Task-Specific Fine-Tuning

We evaluate DGR’s performance across models of varying scales to establish a scaling law and explore fine-tuning as an efficiency enhancement. While large models (e.g., 235B) achieve strong repair results, their inference cost and hardware requirements pose practical

barriers for routine industrial use. This motivates our investigation into whether task-specific fine-tuning can distill repair capabilities into smaller, more deployment-efficient models. The evaluation of the fine-tuned models employs 5-fold cross-validation to ensure the stability and reliability of the results, with all outcomes averaged over three runs. (Table 5).

Table 5. Performance of base and fine-tuned models.

Method	Smell Score	Percentage
Original	3105	100.00%
Base Models		
DeepSeek-R1-14B	1242	40.00%
DeepSeek-R1-32B	1167	37.58%
Qwen3-0.6B	3105	100.00%
Qwen3-8B	1569	50.54%
Qwen3-32B	1487	47.90%
Qwen3-235B	1443	46.47%
Fine-Tuned Models (0.6B)		
Qwen3-0.6B (235B)	1547	49.82%
Qwen3-0.6B (235B + Parfum)	1455	46.86%

Performance improves with model scale within the Qwen3 family: the 235B model performs best (46.47% of original), followed by 32B (47.90%) and 8B (50.54%). The base 0.6B model shows no improvement, indicating its limited capability for this task without fine-tuning.

To examine whether this trend generalizes beyond a single model family, we additionally evaluated DeepSeek-R1 models under the same guided pipeline. As shown in Table 5, DeepSeek-R1-14B and DeepSeek-R1-32B achieve smell scores of 1242 (40.00%) and 1167 (37.58%), respectively. This confirms that DGR’s effectiveness is not tied to a specific model family.

To bridge the capability gap of small models, we constructed two fine-tuning datasets: one containing only self-generated repair examples from the 235B model (Pure-235B), and the other augmented with high-quality repair pairs from the rule-based tool Parfum (235B + Parfum). We then fine-tuned the 0.6B model on these datasets.

After fine-tuning, the 0.6B model shows remarkable gains. The model trained on the 235B-only dataset achieves a score of 49.82%, comparable to the base 8B model. More importantly, the model trained on the combined (235B + Parfum) dataset achieves 46.86%, a performance competitive with the base 32B model. This indicates that incorporating high-quality, rule-based examples enhances fine-tuning effectiveness.

These findings demonstrate that task-specific fine-tuning, especially with a hybrid dataset, offers a practical path to deploy effective Dockerfile repair in resource-constrained environments.

9. Threats to Validity

Internal validity. The non-determinism of LLM repair was mitigated by three repeated trials with a low temperature (0.3) and reporting average results. All builds were performed in an isolated environment with pinned dependencies; failed builds were retried to exclude transient issues. Baseline tools were used with their default configurations and verified for correct operation.

External validity. Our Dockerfile samples were drawn from high-star, active GitHub projects, ensuring real-world relevance, but may not fully represent private repositories or

niche domains. The DGR framework is designed to be general, but its performance metrics depend on the Qwen model series and may vary with other LLMs (e.g., GPT, Llama).

Construct validity. Our core metrics (Smell Count, Smell Score) rely entirely on Hadolint, a community-standard linter. While Hadolint’s rule set and severity weightings represent a particular perspective on best practices, using this widely adopted tool ensures objective evaluation and comparability. This creates a potential evaluation bias because Hadolint is also used in DGR’s detection phase. To mitigate this concern, we report build success rate as an independent practical metric and analyze build failures separately. Nevertheless, Hadolint reflects only one operationalization of Dockerfile quality, and future work should include complementary linters and human assessment.

Conclusion validity. Formal statistical significance tests (e.g., Wilcoxon) were not performed. However, the reported improvements are substantial and consistent across all metrics. The use of multiple metrics does not bias the results, as DGR shows a leading trend across all of them.

10. Conclusion and Future Work

This paper proposed the Detect–Guide–Repair (DGR) framework for automated Docker smell repair, combining rule-based detection with LLM-guided repair. Evaluated on 417 real-world Dockerfiles, DGR reduced smells to 44.68% of the original count and achieved a build success rate of 89.20%, outperforming both rule-based and pure LLM baselines. Our analysis further identified three effective enhancement pathways: hybrid rule–LLM repair, automated error correction, and task-specific fine-tuning of small models.

For future work, we plan to extend DGR to other infrastructure-as-code artifacts (e.g., Kubernetes manifests, CI/CD pipelines), develop dynamic strategies for selecting between rule-based and LLM-based repair, and distill large-model repair capabilities into efficient small models for practical deployment.

Author Contributions: Conceptualization, C.Z., H.W. and J.Z.; Methodology, C.Z., H.W. and J.Z.; Software, C.Z. and J.Z.; Validation, C.Z. and J.Z.; Formal analysis, C.Z., H.W. and J.Z.; Investigation, H.W., Z.G. and L.W.; Resources, Z.G. and L.W.; Data curation, Z.G. and L.W.; Writing—original draft, C.Z. and H.W.; Writing—review & editing, C.Z. and H.W.; Visualization, C.Z.; Supervision, H.W., Z.G. and L.W.; Project administration, Z.G. and L.W.; Funding acquisition, Z.G. and L.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Shenzhen Science and Technology Program, grant number KJZD20240903103811016; the Major Key Project of PCL (Pengcheng Laboratory), grant number PCL2024A05; and the National Natural Science Foundation of China, grant numbers 62572136 and W2511073.

Data Availability Statement: The dataset and source code supporting the reported results are publicly available at https://github.com/three-star-potato/docker_smell_emse (accessed on 2 July 2026).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Docker Documentation. Dockerfile Best Practices. 2025. Available online: <https://docs.docker.com/build/building/best-practices/> (accessed on 25 July 2024).
2. Durieux, T. Empirical study of the docker smells impact on the image size. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024; pp. 1–12.

3. Bui, Q.C.; Laukötter, M.; Scandariato, R. Dockercleaner: Automatic repair of security smells in dockerfiles. In Proceedings of the 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), Bogotá, Colombia, 1–6 October 2023; pp. 160–170.
4. Henkel, J.; Bird, C.; Lahiri, S.K.; Reps, T. Learning from, understanding, and supporting devops artifacts for docker. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, Republic of Korea, 27 June–19 July 2020; pp. 38–49.
5. Eng, K.C.; Hindle, A. Revisiting Dockerfiles in Open Source Software over Time. In Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), Madrid, Spain, 17–19 May 2021; pp. 449–459. [\[CrossRef\]](#)
6. Wu, Y.; Liu, Y.; Wang, T.; Wang, H. Characterizing the Occurrence of Dockerfile Smells in Open-Source Software: An Empirical Study. *IEEE Access* **2020**, *8*, 34127–34139. [\[CrossRef\]](#)
7. Zhou, Y.; Zhan, W.; Li, Z.; Han, T.; Chen, T.; Gall, H. DRIVE: Dockerfile Rule Mining and Violation Detection. *ACM Trans. Softw. Eng. Methodol.* **2024**, *33*, 1–23. [\[CrossRef\]](#)
8. Lin, C.; Nadi, S.; Khazaei, H. A Large-scale Data Set and an Empirical Study of Docker Images Hosted on Docker Hub. In Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), Adelaide, SA, Australia, 28 September–2 October 2020; pp. 371–381. [\[CrossRef\]](#)
9. Rosa, G.; Scalabrino, S.; Robles, G.; Oliveto, R. 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, Lisbon, Portugal, 15–16 April 2024; pp. 231–241. [\[CrossRef\]](#)
10. Docker Community. Docker-Bench-Security: Script That Checks for Best Practices in Docker Deployments. Available online: <https://github.com/docker/docker-bench-security> (accessed on 12 June 2025).
11. Replicatedhq. Dockerfilelint: An Opinionated Dockerfile Linter. Available online: <https://github.com/replicatedhq/dockerfilelint> (accessed on 12 June 2025).
12. Tech, G. Dockle: Container Image Linter for Security. Available online: <https://github.com/goodwithtech/dockle> (accessed on 12 June 2025).
13. Advaitpatel. AI-Powered Docker Security Analyzer. Available online: <https://github.com/advaitpatel/DockSec> (accessed on 12 June 2025).
14. Hadolint. Hadolint: Dockerfile Linter Validate Inline Bash Written in Haskell. Available online: <https://github.com/Hadolint/Hadolint> (accessed on 12 June 2025).
15. Rosa, G.; Scalabrino, S.; Oliveto, R. Fixing Dockerfile Smells: An Empirical Study. *Empir. Softw. Eng.* **2022**, *29*, 108. [\[CrossRef\]](#)
16. Henkel, J.; Silva, D.; Teixeira, L.; d’Amorim, M.; Reps, T. Shipwright: A Human-in-the-Loop System for Dockerfile Repair. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Madrid, Spain, 25–28 May 2021; pp. 198–199. [\[CrossRef\]](#)
17. Bouzenia, I.; Devanbu, P.; Pradel, M. Repairagent: An autonomous, llm-based agent for program repair. *arXiv* **2024**, arXiv:2403.17134. <https://arxiv.org/abs/2403.17134>.
18. Huang, K.; Zhang, J.; Bao, X.; Wang, X.; Liu, Y. Comprehensive Fine-Tuning Large Language Models of Code for Automated Program Repair. *IEEE Trans. Softw. Eng.* **2025**, *51*, 904–928. [\[CrossRef\]](#)
19. Deligiannis, P.; Lal, A.; Mehrotra, N.; Poddar, R.; Rastogi, A. RustAssistant: Using LLMs to fix compilation errors in Rust code. In Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), Ottawa, ON, Canada, 27 April–3 May 2025; pp. 3097–3109. [\[CrossRef\]](#)
20. Shabani, T.; Nashid, N.; Alian, P.; Mesbah, A. Dockerfile Flakiness: Characterization and Repair. In Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), Ottawa, ON, Canada, 26 April–6 May 2025; pp. 1793–1805. [\[CrossRef\]](#)
21. Ye, Z.; Huynh, T.; Le, M.; Babar, M.A. LLMSecConfig: An LLM-Based Approach for Fixing Software Container Misconfigurations. In Proceedings of the 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), Ottawa, ON, Canada, 28–29 April 2025; pp. 629–641. [\[CrossRef\]](#)
22. Ksontini, E.; Mastouri, M.; Khalsi, R.; Kessentini, W. Refactoring for Dockerfile Quality: A Dive into Developer Practices and Automation Potential. In Proceedings of the 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), Ottawa, ON, Canada, 28–29 April 2025; pp. 788–800. [\[CrossRef\]](#)
23. Liu, P.; Ji, S.; Fu, L.; Lu, K.; Zhang, X.; Lee, W.H.; Lu, T.; Chen, W.; Beyah, R. Understanding the security risks of Docker Hub. In *Computer Security—ESORICS 2020, Guildford, UK, 14–18 September 2020*; Springer: Cham, Switzerland, 2020; pp. 257–276.
24. Zhu, Z.; Chen, T.; Liu, C.; Liu, H.; Song, Q.; Xu, Z.; Liu, Y. Doctor: Optimizing Container Rebuild Efficiency by Instruction Re-orchestration. *Proc. ACM Softw. Eng.* **2025**, *2*, 1–23. [\[CrossRef\]](#)

25. Dabic, O.; Aghajani, E.; Bavota, G. Sampling projects in GitHub for MSR studies. In Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), Madrid, Spain, 17–19 May 2021; pp. 560–564.
26. Yang, A.; Li, A.; Yang, B.; Zhang, B.; Hui, B.; Zheng, B.; Yu, B.; Gao, C.; Huang, C.; Lv, C.; et al. Qwen3 Technical Report. 2025. Available online: <https://arxiv.org/abs/2505.09388> (accessed on 25 May 2026).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.