

Refactoring for Dockerfile Quality: A Dive into Developer Practices and Automation Potential

Emna Ksontini

emna@umich.edu

University of Michigan - Flint
USA

Rania Khalsi

rkhalsi@umich.edu

University of Michigan - Flint
USA

Meriem Mastouri

meriemm@umich.edu

University of Michigan - Flint
USA

Wael Kessentini

wkessent@depaul.edu

DePaul University
USA

Abstract—Docker, the industry standard for packaging and deploying applications, leverages Infrastructure as Code (IaC) principles to facilitate the creation of images through Dockerfiles. However, maintaining Dockerfiles presents significant challenges. Refactoring, in particular, is often a manual and complex process.

This paper explores the utility and practicality of automating Dockerfile refactoring using 600 Dockerfiles from 358 open-source projects. Our study reveals that Dockerfile image size and build duration tend to increase as projects evolve, with developers often postponing refactoring efforts until later stages in the development cycle. This trend motivates the automation of refactoring. To achieve this, we leverage In Context Learning (ICL) along with a score-based demonstration selection strategy. Our approach leads to an average reduction of 32% in image size and a 6% decrease in build duration, with improvements in understandability and maintainability observed in 77% and 91% of cases, respectively. Additionally, our analysis shows that automated refactoring reduces Dockerfile image size by 2x compared to manual refactoring and 10x compared to smell-fixing tools like PARFUM.

This work establishes a foundation for automating Dockerfile refactoring, indicating that such automation could become a standard practice within CI/CD pipelines to enhance Dockerfile quality throughout every step of the software development lifecycle.

Index Terms—Refactoring, In-Context Learning, Software Engineering, LLM, Docker Refactoring, IaC.

I. INTRODUCTION

Docker has emerged as the de facto industry standard, offering a platform for creating, deploying, and managing containers [1]. The contents of a Docker container are defined by Dockerfile declarations, which specify the instructions and their execution order. Often found in source code repositories, these files facilitate the building of images that can then be run as containers, thereby bringing the hosted software to life in its execution environment [2].

Despite being the backbone of containerization, Dockerfiles are often plagued with problems such as smells [3]. In this context, several works have been proposed to detect and fix them [4–6]. However, most solutions focus on bash commands, which are typically embedded within specific instructions

in Dockerfiles, mainly the RUN instruction. Common bash smells include the installation of unnecessary packages and missing version pinning [4, 7]. While bash-based solutions are useful, they fall short in addressing the broader structure of Dockerfiles, which are not limited to the RUN instruction. Instead, Dockerfiles include various instructions and stages that together form the final image.

Developers face challenges beyond managing shell scripts in Dockerfiles, such as reducing technical debt, as defined in [8], which affects all instructions and stages, compromising overall quality. For instance, a key challenge is maintaining small **image sizes**. Each instruction within a Dockerfile adds a layer to the final image, potentially leading to bloated containers if not managed carefully. Choosing the appropriate base image and effectively utilizing multi-stage builds can mitigate this issue, but achieving the right balance requires considerable expertise and effort [3, 9]. **Maintaining** Dockerfiles is another challenge, with common bugs and unexpected behaviors often arising, especially when the base image tag is not specified correctly. Additionally, some developers, unaware of the availability of official and trusted images, manually add dependencies, which can lead to inconsistencies and complicate future updates [10]. **Build duration** is influenced by several factors, including the order of instructions, the complexity of multi-step commands, and the number of instructions. Poorly optimized sequences and complex instructions can lead to high build durations and incur high costs, complicating the workflow [11]. As a project progresses through its development stages, the content of the Dockerfile may be revised many times [12], which can affect **understandability** and complicate subsequent updates.

Current refactoring tools designed for languages like Java, JavaScript, C, and Python [13] are inadequate for Dockerfiles due to the distinct nature of IaC and their declarative syntax. Furthermore, Dockerfiles are used to containerize applications across a wide array of programming languages (e.g., Java, PHP, Go) and domains (e.g., machine learning, web development), which requires refactoring efforts that

extend beyond mere structural changes. Effective refactoring often demands domain expertise in selecting appropriate base images, specifying correct tags, and adjusting dependencies. This complicates the application of search-based techniques in this context. At the same time, using deep learning (DL) approaches poses major challenges due to its data-intensive nature. Even with access to the largest Dockerfile collection available in the literature, DL models fell short in capturing the intricate patterns and variability unique to Dockerfiles [14].

To address the data shortage problem, we propose leveraging Large Language Models (LLMs) [15], which have recently emerged as powerful tools for various software engineering tasks [16]. These models are pre-trained on vast datasets, including billions of code repositories, enabling them to capture substantial domain knowledge. We hypothesize that this exposure allows LLMs to generate Dockerfile refactorings that closely align with common practices and are likely to be adopted by developers.

To optimize LLM performance for this purpose, we employ ICL. Recent studies show that ICL, as opposed to straightforward prompts, effectively leverages the domain knowledge embedded within LLMs by aligning input formats with those used in pre-training [17, 18]. This strategy aligns with recent industry trends: Docker recently launched labs with a generative AI assistant (based on the GPT-4 model) to aid Dockerfile creation, and they reported that prompts lacking best practices led to lower-quality Dockerfiles [19].

In this paper, we introduce a novel approach that harnesses the capabilities of ICL to automate Dockerfile refactoring. Our methodology involves four key components: (1) an analysis of how developers currently perform Dockerfile refactoring within open-source projects, aiming to assess the utility and potential benefits of automating these tasks; (2) an evaluation of ICL effectiveness in automating refactoring, using a novel score-based selection method for demonstration examples, with a focus on optimizing image size, build duration, understandability, and maintainability; (3) a comparative analysis of automated refactoring, manual refactoring, and smell-fixing tools with respect to their effectiveness in enhancing Dockerfile quality, and (4) a thorough examination of the reasons behind build failures caused by refactoring.

To the best of our knowledge, this is the first study on Dockerfile refactoring automation, revealing several findings:

- Developers predominantly engage in Dockerfile refactoring during the mid to late stages of project development, often addressing accumulated technical debt.
- In general, the effectiveness of ICL in automating Dockerfile refactoring significantly improves with an increasing number of demonstrations.
- Automated refactoring outperforms both manual refactoring and smell repair tools, achieving greater reductions in Docker image size and build duration, while also improving the understandability and maintainability.
- The primary causes of build failures in both automated and manual Dockerfile refactorings are related to build

context errors and dependency management.

Replication Package. All materials are available in [20].

II. RELATED WORK

A. Docker Quality Issues

Docker's documentation offers best practices [21] for addressing Dockerfile smells. However, developers' adherence to these guidelines is inconsistent. These smells, akin to traditional configuration code smells [22], indicate design flaws that, although not bugs, negatively affect Docker images.

In [4], authors have categorized Dockerfile smells into two primary types: *DL*-smells, which breach Docker's official best practices, and *SC*-smells, which violate basic shell script conventions. Their empirical study revealed that most open-source projects exhibited Dockerfile smells, with *DL*-smells occurring more frequently than *SC*-smells. Similarly, Durieux et al. [3] have investigated how shell smells in Dockerfiles increase image size, affecting distribution efficiency and scalability.

Henkel et al. [23] have developed Binnacle to mine Bash-related smells from over 178,000 Dockerfiles. Xu et al. [24] have introduced *TF*-smells, showing how the risky practice of leaving temporary files in Docker images during the build process can inflate image size and complicate distribution.

Beyond detection, some studies have explored the automation of Dockerfile smell repairs. For example, *PARFUM* [6] detects and automatically repairs Dockerfile smells, evaluating repair effectiveness in terms of build failure and image size. Rosa et al. [7] have evaluated Hadolint writing practices by experts and ranked 26 additional Dockerfile smells, offering guidance for high-quality Dockerfile writing.

Other studies, such as [25], used linear regression to analyze Dockerfile quality and project characteristics, revealing statistical correlations. Cito et al. [26] found that missing version pinning smell in Dockerfiles often causes build issues, leading to non-reproducible builds. In [27], a human-in-the-loop system using a modified BERT model has been developed for repairing Dockerfiles, showing potential for automating repairs and boosting build success.

While most studies focus on addressing Bash smells within the RUN instruction, Dockerfiles encompass a broader range of instructions and stages that can impact overall quality. Addressing these broader quality issues is crucial, as they often signal technical debt in Docker projects. Such debts can manifest as excessively large images, which in turn lead to slower deployment times and increased resource consumption. Refactoring these elements is essential to improve performance and efficiency in Docker-based environments [8].

In [28], DRMiner has been introduced for identifying and analyzing Dockerfile refactorings, providing semantics-aware static analysis. Although this tool indicates future potential for automated Dockerfile refactoring, to the best of our knowledge, no tool currently achieves full automation. Similarly, [29] have suggested methods to improve Dockerfile quality and automate the generation of high-quality Dockerfiles. These studies shift focus from detecting shell smells to addressing

border issues through potential automation, enhancing Dockerfile quality.

In this paper, we fully automate the refactoring of Dockerfiles to reduce technical debt and improve their quality.

B. Refactoring Recommendation and Automation

Several research efforts have aimed to assist or automate code refactoring using various techniques. Despite the availability of automated tools, studies reveal that refactoring is often performed manually due to issues like tool integration and lack of support for real-world scenarios [30].

Search-based techniques [31] treat software refactoring as an optimization problem aimed at enhancing system design quality using software metrics. Ouni et al. [32] developed a multi-objective formulation to address code smells, while Alizadeh et al. [33] employed NSGA-II for dynamically and interactively recommending refactoring solutions.

ML techniques have also been widely used for recommending refactorings. Nyamawe et al. [34] utilized classifiers like LR, CNN, and SVM to suggest refactorings based on historical data and detected code smells. Sagar et al. [35] created a dataset of five refactoring types using 800 open-source Java projects, applying RF, SVM, and LR for identification.

Recent studies highlight in-context learning's potential to enhance LLM performance in software engineering tasks [36]. Zhang et al. [37] showed LLMs have promising bug-fixing abilities. Madaan et al. [38] used Codex, to improve code readability, while Shirafuji et al. [39] demonstrated GPT-3.5's effectiveness in refactoring Python programs, significantly reducing complexity and code lines.

Despite these advancements, there is a notable gap in tools specifically designed for IaC, particularly for Docker. Srivatsa et al. [40] emphasized the potential of LLMs in generating IaC configurations, suggesting automation can make these processes more accessible.

To our knowledge, no studies have automated Dockerfile refactoring or applied ICL to improve Docker quality. This research investigates automated refactoring, comparing its effectiveness to manual refactoring and smell-fixing tools.

III. STUDY DESIGN

A. Research Questions

Our study investigates the practicality and efficiency of automating Dockerfile refactoring. We address the following research questions, summarized in Figure 1.

RQ1: To what extent do developers' refactoring habits correlate with changes in Docker image size and build duration?

Getting insights into how Dockerfiles evolve in open-source projects is crucial for understanding the benefits of automating their refactoring. Key aspects include identifying when developers refactor Dockerfiles, the frequency of these refactorings, and the impact of Dockerfile changes on performance metrics such as build duration and image size. Analyzing these elements can highlight the time-saving opportunities and performance improvements that automation can offer.

To address RQ1, we perform a quantitative analysis of 75 Dockerfiles from 75 open-source GitHub projects, illustrated in Figure 1 (yellow arrows). For the commit history of each Dockerfile, we identify instances of refactoring using DRMiner [28]. We then map out the lifecycle stages during which these refactoring actions occurred. Given the varying number of commits and lifetimes of the studied projects, each project's lifecycle is segmented into ten equal phases: the first 10% of the total commits, the next 10%, and so on. In this context, we exclude projects with less than ten commits.

This allows for determining when developers typically start refactoring and during which stage it happens most frequently.

Beyond identifying refactoring instances, we automatically build each Dockerfile for every commit in its history. If successful, we measure the build duration and image size to study their average increase during the Dockerfile's evolution.

RQ2: How effective is ICL in automating Dockerfile refactoring using zero-shot, one-shot, and few-shot learning approaches?

This research question evaluates the effectiveness and reliability of ICL for automating Dockerfile refactoring, focusing on how zero-shot, one-shot, and few-shot approaches impact model performance.

To answer RQ2, we utilize a dataset comprising 202 refactored Dockerfiles from 120 unique open-source projects. As illustrated in Figure 1 (green arrows), we start by prompting the LLM with the pre-refactoring version of each Dockerfile (V_{Before_Test}), using a zero-shot approach. Next, we examine the effectiveness of one-shot and few-shot learning by including one or more refactoring demonstrations in the prompt. To select these examples, We develop a customized selection strategy that employs score-based ranking to identify the most relevant refactoring demonstrations from a training set of 398 Dockerfile refactoring commits across 238 projects. Moreover, during few-shot learning, we evaluate the impact of increasing the number of demonstrations on the model's performance.

During these experiments, each generated Dockerfile refactoring undergoes a behavioral assessment to ensure its functional behavior remains unchanged. Finally, we evaluate effectiveness using several key metrics, including build failure rates, maintainability, understandability, image size, and build duration.

RQ3: How effective is automated Dockerfile refactoring in improving quality, compared to manual refactoring and smell-detection tools?

In this RQ, we aim to determine how well Dockerfile refactorings automation compare to those performed by developers. This evaluation will offer insights into the practicality of automated refactoring in real-world software development.

Moreover, we found that existing techniques, such as PARFUM [6], an automated tool for repairing Dockerfile bash-related smells, have demonstrated measurable improvements in Dockerfile quality, notably in reducing image size [3]. We aim to contrast the efficacy of improvements achieved through automated refactoring with those driven purely by smell fixing. We compare with PARFUM since it is a recent work offering

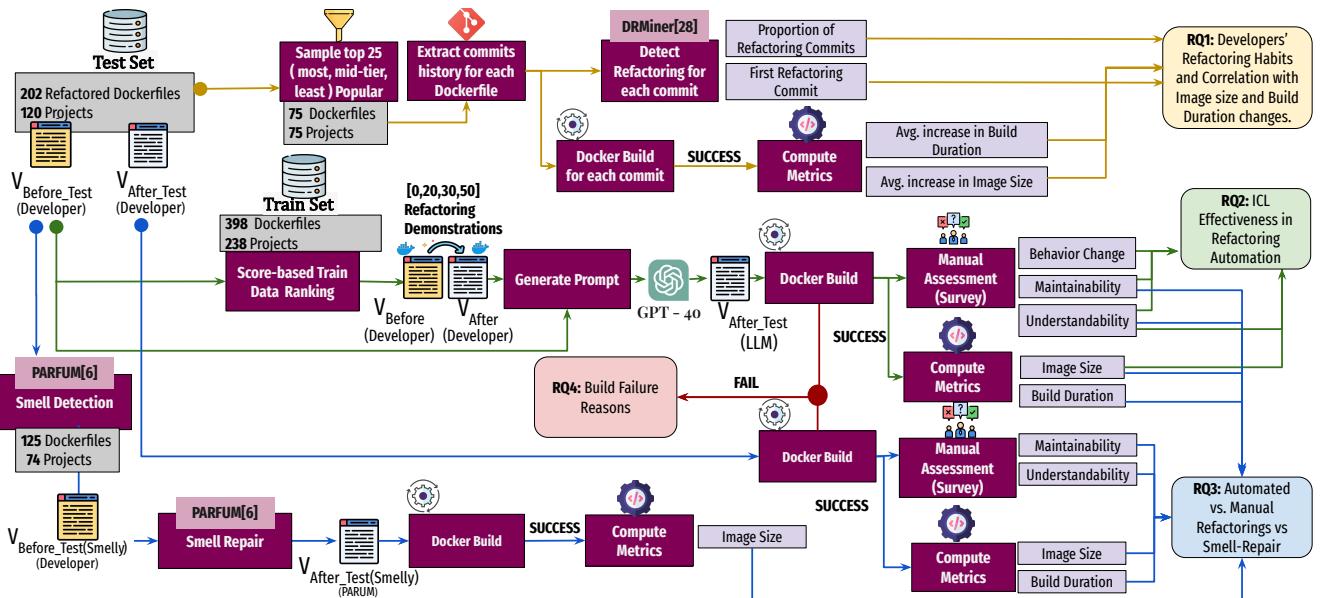


Figure 1: Approach overview and addressed RQs (RQ1: yellow arrows, RQ2: green arrows, RQ3: blue arrows, and RQ4: red arrows)

automated repairs or Dockerfile smells. In contrast, other techniques, such as Shipwright [27], Hadolint [41], and Binnacle [23], focus on detecting error patterns or smells and require manual intervention for their operation.

To answer **RQ3**, we use the results from the best-performing ICL approach identified in **RQ2**. We first compare automatically refactored Dockerfiles, denoted ($V_{\text{After_Test}}(\text{LLM})$), with their human-refactored versions, denoted ($V_{\text{After_Test}}(\text{Developer})$), as illustrated in Figure 1 (blue arrows). This comparison is based on the same effectiveness aspects used in **RQ2**: build failure, image size, build duration, understandability, and maintainability. We then contrast the results of automated refactoring with those achieved solely through smell correction, specifically noted as ($V_{\text{After_Test(smelly)}}(\text{PARFUM})$), focusing on image size as a core outcome measure. For a fair comparison with PARFUM, we restricted this analysis to Dockerfiles identified as “smelly”, representing 125 instances from our test set.

RQ4: What causes Build Failures after refactoring, and how do these reasons differ between automated and manual approaches?

In **RQ2** and **RQ3**, it was observed that while the ICL achieved commendable improvements in image size, build duration, understandability, and maintainability, there was a considerable rate of build failures in the generated refactored Dockerfiles. Interestingly, we observed a similar rate of build failures in the Dockerfiles refactored by human developers. In this research question, we aim to understand the reasons behind these build failures. By uncovering these reasons, the aim is to identify further hidden factors related to Dockerfile refactoring practices and improve the automation process.

To answer **RQ4**, we conduct a qualitative analysis of build failures caused by both ICL and human developers over a set of 182 Dockerfile refactoring cases. The steps for this analysis

are illustrated in Figure 1 (red arrows).

B. Refactored Dockerfiles Data Collection

To construct our dataset, we gather refactored Dockerfiles from the commit history of Docker projects, focusing on commits created primarily for refactoring. The parent versions of these Dockerfiles are assumed to have quality problems, as evidenced by the developers’ choice to refactor them. This makes them relevant candidates for assessing the effectiveness of automated refactoring processes. The detailed extraction and filtering procedures figure is available in [20].

To begin with, we use the Google BigQuery GitHub Public Dataset [42], released in 2016, to source our data. We focus on the most recent snapshot available as of March 10, 2024, and target commit messages containing terms referring to self-affirmed refactoring patterns such as “*refactor.**”, “*fix.**”, “*improve.**”, following findings from AlOmar et al. [43]. In addition, we ensure that these commits mention the word “*Dockerfile*”. This query yields 3,046 Dockerfiles from 1634 commits across 1,293 projects, representing potential pre- and post-refactoring versions of a Dockerfile (V_{before} and V_{after}).

Next, we perform an automated build, and Dockerfiles that fail to build in their pre-refactored state are discarded, leaving us with 650 successful Dockerfile pairs from 381 projects. This ensures that we are working with functional Dockerfiles before applying automation.

We process the dataset further using DRMiner [28], a tool that detects specific refactoring actions performed on Dockerfiles. While DRMiner is designed to identify 12 refactoring types, the “*Move Stage*” and “*Extract Run Instruction*” types are only present in two Dockerfiles in our dataset. Therefore, we decide to omit these two types and focus on the remaining 10 refactoring types. After this step, we are left with 641 Dockerfiles from 375 projects.

To ensure that the dataset only includes refactoring changes, three authors examined the Dockerfile pairs and excluded files with functional behavior changes. Functional behavior refers to Dockerfile elements that affect the application’s runtime behavior, namely application files (*COPY/ADD*) and startup commands (*ENTRYPOINT/CMD*). Following this step, we retained 600 Dockerfiles from 358 projects.

At this stage, we split the dataset into training (75%) and test (25%) sets using stratified sampling to ensure balanced representation of refactoring types. To avoid having Dockerfiles from the same projects in both sets, we moved about 2% (12 Dockerfiles) to the test set, resulting in 27% (162 Dockerfiles) in the test set and 73% (438 Dockerfiles) in the training set. Finally, we verify that all Dockerfiles in the training set were successfully built post-refactoring. This additional verification step resulted in a further 7% (40 Dockerfiles) of the data being removed from the training set and moved to the test set due to build failures. The final dataset distribution comprises 398 Dockerfile revisions (approximately 66%) from 238 unique projects in the **training set** and 202 Dockerfile revisions from 120 unique projects (approximately 34%) in the **test set**.

C. The Prompt Template for Refactoring Automation

Our prompt is defined as: $P = \{N + RD + V_{\text{Before_Test}}\}$ where N is a natural language template specifying the task description and the definition of all possible refactoring actions [8]. $RD = \{V_{\text{Before}_i}, V_{\text{After}_i}, R_i\}_{i=1}^n$ is a set of refactoring demonstrations composed of the input Dockerfile V_{Before_i} , the desired output Dockerfile V_{After_i} , and the refactoring actions applied (R_i). $V_{\text{Before_Test}}$ is a Dockerfile for testing. Specifically, if $n = 0$, indicating no refactoring demonstration, the setting is known as *zero-shot learning*; if $n = 1$, indicating only one refactoring demonstration, the setting is known as *one-shot learning*; and *few-shot learning* applies when there are multiple refactoring demonstrations. In addition, there is a constraint that $\text{size}(P) \leq \text{context_window}$, which means that the prompt should fit within the token limit of the language model. A figure detailing the prompt can be found in [20].

D. Refactoring Demonstration Retrieval

Existing research demonstrates that ICL achieves better performance in code intelligence tasks with instance-specific demonstrations tailored to test inputs rather than task-level ones [17, 36, 44]. Drawing on this finding, in RQ2 we develop a score-based demonstration selection strategy based on technical debts defined in [8]. As illustrated in Figure 1. Given a test Dockerfile $V_{\text{Before_Test}}$, we identify the most relevant dockerfile refactoring demonstrations RD_i using the following formula:

$$\begin{aligned} \text{score}(RD_i) = & 0.2 \times \text{Textual_similarity}(V_{\text{Before}_i}, V_{\text{Before_Test}}) + \\ & 0.2 \times \text{understandability_score}(V_{\text{Before}_i}, V_{\text{After}_i}) + \\ & 0.2 \times \text{maintainability_score}(V_{\text{Before}_i}, V_{\text{After}_i}) + \\ & 0.2 \times \text{image_size_score}(V_{\text{Before}_i}, V_{\text{After}_i}) + \\ & 0.2 \times \text{build_duration_score}(V_{\text{Before}_i}, V_{\text{After}_i}) \end{aligned}$$

The components of the score are defined as follows:

- **Textual_similarity**: Calculated using BM-25 [45].
- **Understandability (or Maintainability) score**: Assigned a value of 1 if the understandability (or maintainability) of V_{After_i} is greater than V_{Before_i} , 0 if they are the same, and -1 if it is worse.
- **Build duration (or image size) score**: Calculated as $1 - \frac{\text{build_duration(or image_size)}(V_{\text{After}_i})}{\text{build_duration(or image_size)}(V_{\text{Before}_i})}$, quantifying the improvement in build duration (or image size).

We use an automated script to compute the image size (MB) and build duration (s). To ensure consistency, we measure the build duration over three runs and use the average. For maintainability and understandability, we employ a manual assessment performed by three authors of the paper. Each Dockerfile was labeled three times, with assessments deemed valid if at least two evaluators agreed.

For each query, we compute the score over the entire corpus of demonstrations, rank the train corpus based on this score, and select the top n based on the number of shots. For instance, we select the top example for one-shot, the top 20 examples for 20-shot, and so on. When adding the demonstrations to the prompt, we order them in ascending order, meaning the one with the highest score will be closest to the query Dockerfile. This ordering follows the findings by Gao et al. [36].

This scoring strategy returns demonstrations that closely match the query Dockerfile and exhibit the greatest overall quality improvements. By focusing on content similarity, we ensure that the model understands the specific needs of different Dockerfiles in terms of applications and commands. Moreover, prioritizing quality improvements enables the model to learn optimal practices from the best examples.

E. Evaluation Metrics

To assess the effectiveness of manual and automated refactorings in RQ2 and RQ3, we also rely on the technical debt outlined in [8]. This involves measuring the image size and build duration for each Dockerfile version: original, manually refactored, and automatically refactored. For maintainability and understandability, we use the scoring system detailed in subsection III-D.

Six industry developers, each with about two years of Dockerfile experience, evaluated maintainability and understandability in an industry-sponsored collaboration. To ensure unbiased evaluations, developers were provided with pairs of Dockerfiles (original and refactored) without being informed which was the refactored version. They were asked to score the improvements in maintainability and understandability. For cases involving LLM-generated refactorings, developers were asked to report any changes in the Dockerfile’s behavior, ensuring that any functional discrepancies were identified (manual refactoring cases had been pre-screened for behavioral changes during data collection).

Participants completed a pre-study survey on their programming experience and company roles. To minimize bias, they received a two-hour lecture on Docker quality assessment and refactoring, covering general and specific Dockerfile practices.

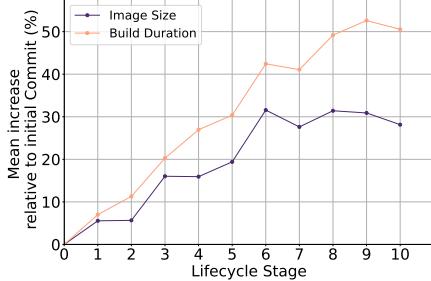


Figure 2: Mean Image Size & Build Duration Increase Over Project Lifecycle

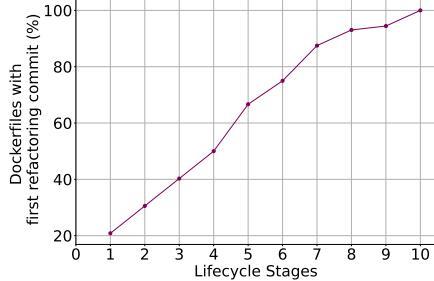


Figure 3: Cumulative percentage of Dockerfiles with first refactoring commit.



Figure 4: Mean proportion of refactoring commits across project lifecycles.

For result annotation, we used the Kappa cross-validation process. Each participant evaluated 300 Dockerfiles, with each pair assessed by three developers. An assessment was considered valid if at least two evaluators agreed, achieving an overall Cohen’s kappa of 0.84. Detailed information about the participants and evaluated Dockerfiles is available in our replication package [20].

F. Experiment Settings

The rationale behind in-context learning is that LLMs, having been trained on vast corpora, absorb substantial domain knowledge, enabling them to generalize well to new tasks without the need for fine-tuning [46].

Among these LLMs, GPT-4 (Generative Pre-trained Transformer 4) has shown remarkable performance not only in conventional code-related tasks [47] but also in IaC scenarios [48]. Consequently, we have selected GPT-4o (an optimized variant of GPT-4) as the primary model for our experimental framework. Regarding the hyperparameters of the model’s API, and in alignment with previous studies [17, 36], we set the temperature to 0 to obtain deterministic outputs.

The context window of GPT-4o, which accommodates up to 128k tokens, imposes practical limits on the number of demonstrations we can use. Given that the largest Dockerfiles in our dataset are around 1200 tokens, and the combined size of the task description and refactoring action definitions is 434 tokens, each demonstration (involving two Dockerfiles plus refactoring actions) totals 2447 tokens. Based on these calculations, we determine that we can safely include up to 50 refactoring examples without exceeding the context window.

To explore different sizes of input examples, our experimental setup includes zero-shot, one-shot, and few-shot settings. For the few-shot settings, we incorporate 20, 30, and 50 refactoring demonstrations. We select 20 and 30 as intermediate steps to balance between the minimum (1-shot) and maximum (50-shot) feasible demonstrations, allowing us to observe the model’s performance across this range.

All experiments were conducted on an 8-Core workstation equipped with an Intel(R) Xeon(R) CPU and 64 GB of RAM, operating on Ubuntu 18.04.

IV. RESULTS

A. RQ1: Developers’ refactoring habits and correlation with image size and build duration changes

To address RQ1, we need to build each Dockerfile for every commit within the project. To ensure a balance between representativeness and computational feasibility, we selected a sample of Dockerfiles based on GitHub star counts. This sample includes 25 Dockerfiles each from high-, mid-, and low-popularity projects, representing a variety of languages and domains. In total, we selected 75 Dockerfiles from 75 projects. The demographics of these Dockerfiles are detailed in [20].

We build these Dockerfiles at every commit in the project’s history, covering 1,519 Dockerfile commits. At each commit, we measure the image size and build duration. We note that in instances of build failures, which occurred in 43% (653/1,519) of the cases, we maintain continuity by using the image size and build duration from the next successful commit. In rare cases where the Dockerfile failed to build on the last commit (one case), we use the previous successful commit. We identify three outliers with image size increases of 300-500% between two commits due to entire Dockerfile replacements. After removing these outliers, *our analysis focuses on 72 Dockerfiles, covering a total of 1,459 Dockerfile commits*. Figure 2 illustrates the mean increase in Dockerfile image size and build duration relative to the initial commit across project lifecycle stages.

Dockerfile image size grows initially, peaks mid-project, and then stabilizes. The early stages (1-3) of the project show a steady increase in Docker image size, with an average growth rate of 15% by stage 3. This upward trend continues into the middle stages (4-6), peaking at stage 6 with a 31% increase. In the later stages (7-10), the image size decreases slightly and stabilizes around a 30% increase from the original size.

Dockerfile build duration rises steadily, peaking at the project’s end. Initially (1-3), the build duration increases, reaching around 20% by stage 3. In the mid-stages (4-6) there is a notable increase, with build duration rising to about 41%. In the later stages (7-10), this trend peaks at around 50% by stage 8 and then remains relatively constant towards the end.

Table I: Evaluation of Prompting Techniques and Developer Performance

Prompting Techniques / Developer	Build Success Rate (%)	Image Size			Build Duration			Understandability		Maintainability			
		Improvement Rate (%)	Deterioration Rate (%)	Average Reduction	Total Reduction (GB)	Improvement Rate (%)	Deterioration Rate (%)	Average Reduction	Total Reduction (minutes)	Improvement Rate (%)	Deterioration Rate (%)	Improvement Rate (%)	Deterioration Rate (%)
Zero-shot	38% (77/202)	81% (61/75)	16% (12/75)	129 MB (18%)	9 GB	48% (36/75)	52% (39/75)	-12 s (-38%)	-15 min	93% (70/75)	0% (0/75)	99% (74/75)	0% (0/75)
One-shot	47% (94/202)	75% (68/91)	21% (19/91)	145 MB (25%)	13 GB	59% (54/91)	41% (37/91)	32 s (6%)	49 min	87% (79/91)	0% (0/91)	97% (88/91)	0% (0/91)
20-shot	49% (99/202)	73% (72/99)	22% (22/99)	245 MB (18%)	24 GB	58% (57/99)	42% (42/99)	30 s (-5%)	49 min	80% (79/99)	0% (0/99)	96% (95/99)	0% (0/99)
30-shot	54% (108/202)	81% (88/108)	15% (16/108)	274 MB (25%)	29 GB	59% (64/108)	41% (44/108)	22 s (-1%)	39 min	80% (86/108)	0% (0/108)	95% (103/108)	0% (0/108)
50-shot	63% (128/202)	82% (105/128)	13% (16/128)	322 MB (32%)	35 GB	70% (89/128)	30% (39/128)	24 s (6%)	52 min	77% (98/128)	2% (2/128)	91% (117/128)	1% (1/128)
Developer	47% (94/202)	53% (50/94)	35% (33/94)	79 MB (4%)	7 GB	52% (49/94)	48% (45/94)	-2 s (-41%)	-4 min	56% (53/94)	7% (7/94)	82% (77/94)	6% (6/94)

Finding-1. Docker projects in our dataset shows a mean increase of 30% in image size and 50% in build duration from initial development to project maturity.

In addition to analyzing the evolution of performance metrics, we investigate the temporal patterns of Dockerfile refactoring activities across Dockerfiles commits history. Out of 1,459 commits, 739 involve refactoring actions.

Most developers start refactoring Dockerfiles in the mid to later stages of project development. Figure 3 shows the cumulative percentage of Dockerfiles with first refactoring action across lifecycle stages: In stages (1-3), only 40% (28/72) had initiated refactoring. A significant uptick is observed in middle stage stages (4-7), reaching 89% (64/72) by stage 7.

Refactoring activities are most prevalent during the middle stages of project development, where they constitute more than half of the Dockerfile commits. Figure 4 shows the mean proportion of Dockerfile refactoring commits in relation to all Dockerfile commits at each stage of the lifecycle, highlighting the frequency of refactorings during different stages. In the early stages (1-3), refactoring commits compromise approximately 35% of commits. As the project advances (4-7), there is a notable increase in refactoring, peaking at 58% in stage 5 and remaining above 50%. In the later stages (8-10), the refactoring commits rise to 60% at stage 8, before stabilizing around 45%.

Finding-2. 60% percent of Dockerfiles in our dataset underwent their first refactoring in the mid to later stages of development. During this period, refactoring actions made up over 50% of all commits.

The interplay between Dockerfile refactoring activities and the evolution of image size and build duration reveals insights into Docker project development practices. Our analysis suggests that early project phases often experience rapid increases in Docker image size, which may be driven by a focus on feature expansion and the incorporation of dependencies, with less emphasis on refactoring. This could result in the accumulation of technical debts. As projects mature, particularly during the middle stages, an uptick in refactoring activities coincides with a peak in image size growth, potentially indicating a critical shift toward optimization. Interestingly, while these

refactoring efforts may contribute to the stabilization of image size, they can also coincide with an increase in build duration, suggesting that improvements in image size often entail more complex build procedures. These observations highlight the potential benefits of initiating refactoring efforts early in the project lifecycle and maintaining them consistently to prevent the accumulation of technical debts that could complicate Dockerfile development.

B. RQ2: Effectiveness of ICL in automating Dockerfile refactoring using zero-shot, one-shot, and few-shot approaches

In RQ2, we investigate the use of LLMs to automate the process of Dockerfile refactoring. Our study involved different configurations, specifically zero-shot, one-shot, 20-shot, 30-shot, and 50-shot approaches. The results, detailed in Table I, emphasize the effectiveness of LLM and the impact of the number of demonstrations on the rate of successful builds and the reduction of technical debts.

The build success rate exhibits a positive correlation with the number of refactoring demonstrations provided. In zero-shot, only 38% (77/202) of Dockerfiles were successfully built. This success rate increased to 47% (94/202) in one-shot, 49% (99/202) in 20-shot, 54% (108/202) in 30-shot, and reached the highest at 63% (128/202) in 50-shot.

We conducted a behavior assessment for Dockerfiles that were successfully built in all settings. We also evaluate key metrics—image size, build duration, understandability, and maintainability—using improvement rate, deterioration rate, and average reduction. The improvement rate measures the proportion of Dockerfiles showing positive changes post-refactoring, while the deterioration rate indicates the proportion experiencing negative changes. Average reduction quantifies the magnitude of improvement among all the successfully built Dockerfiles. The behavior assessment revealed a very low incidence of functional changes, with only 5 cases (2 zero-shot, 3 one-shot) where Dockerfile behavior changed post-refactoring. These were excluded from our calculations.

The average reduction rate in image size reaches its highest values with more refactoring demonstrations. When examining image size, the improvement rate was consistently high across most settings, hovering around 81% for zero-shot, one-shot, and 50-shot and 72% and 75% respectively for the one-shot and 20-shot. Regarding average image size reduction, zero-shot achieved an average of 18% (129 MB),

whereas both one-shot and 30-shot settings achieved similar reduction rates of around 25%. The 50-shot setting showed the most significant reduction, with an average of 32% (322 MB), resulting in a total saving of 35 GB across all Dockerfiles.

The Build duration average reduction fluctuates, showing the highest improvement with more refactoring demonstrations. Upon analyzing the build duration, we find that the zero-shot setting has the lowest rate of improvement at 48%. This suggests that over half of the successful builds experienced longer build duration after the refactoring. The one-shot, 20-shot, and 30-shot configurations exhibit similar enhancement, achieving improvement rates of approximately 58-59%. The 50-shot configuration demonstrates the most notable improvement rate, reaching 70%. In terms of the average reduction in build duration, the one-shot and 50-shot settings were the most successful, resulting in average reductions of 6% (32 seconds and 24 seconds, respectively). The total amount of time saved using the 50-shot setting was 52 minutes.

Understandability shows an inverse correlation with the number of refactoring demonstrations. In the zero-shot setting, the understandability improvement rate is 93%, but this rate decreased to 87% in the one-shot setting and further to 77% in the 50-shot setting.

Maintainability shows an inverse correlation with the number of refactoring demonstrations provided, with high overall results across all settings. Maintainability was highest in the zero-shot setting at a rate of 99%. This slightly decreased to 95% in the 30-shot setting and 91% in the 50-shot setting. Despite these reductions, maintainability remained high across all settings.

Finding-3. Increasing refactoring demonstrations to 50-shot improves ICL performance in build success rate (63%), image size reduction (32%), and build duration reduction (6%). However, this negatively affects maintainability and understandability, though the improvement rates remain high at 77% and 91%, respectively.

Furthermore, we study the correlation between different key metrics during the 50-shot setting to understand whether refactorings improve these metrics together or if the improvement of one metric leads to the deterioration of another. Using Spearman correlation [49], we find a moderate negative correlation (-0.60) between image size and build duration, with smaller images often increasing build duration. A strong negative correlation (-0.80) between build duration and both understandability and maintainability suggests shorter builds may reduce clarity and maintainability. Understandability and maintainability are perfectly correlated (1.00), while image size has no effect on them (0.00).The correlation matrix can be found in [20].

Finding-4. Improvements in image size often lead to longer build durations, and reducing build duration tends to decrease understandability and maintainability. Conversely, improvements in understandability directly enhance maintainability.

The distribution of refactoring techniques is nearly consistent across all prompting methods, as illustrated in Figure 5. This consistency is observed in zero-shot, one-shot, and few-shot settings, where techniques such as “Extract Stage,” “Rename Image”, and “Update Image Tag” are notably prevalent, each with frequencies exceeding 15%. These techniques emphasize enhancing image size, maintainability (version control), and understandability (renaming). On the other hand, techniques such as “Inline Run Instruction” and “Sort Instruction” consistently exhibit lower frequencies, generally around 4% - 7%. Notably, the “Inline Stage” refactoring was not performed in any setting.

Finding-5. Regardless of the number of demonstrations, the LLM consistently applied the same refactoring techniques. However, varied evaluation metrics indicate that the specific implementation and choices made (e.g., base image, tag, stages, etc.) during the refactoring process impact its effectiveness.

C. *RQ3: Effectiveness of automated Dockerfile refactoring in improving quality compared to manual refactoring and smell-detection tools.*

While metrics show significant improvements through automation of Dockerfile refactoring, comparing these results with human developers’ work is essential for practical validation. We benchmark against human refactorings using the 50-shot setting, our best performer, to identify strengths and areas for improvement.

We have a manually refactored version of the 202 test Dockerfiles. We compute key metrics for these developer versions, such as build success rate, image size, build duration, maintainability, and understandability, as shown in Table I. Developers achieved a build success rate of 47% (92/202), while the 50-shot setting achieved 63% (128/202).

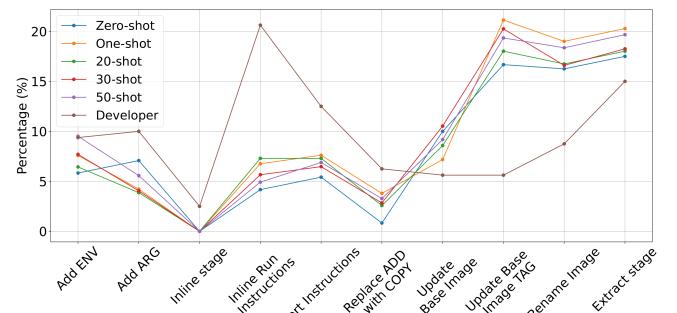


Figure 5: Refactoring Techniques Distribution

To ensure a detailed comparison between automated and manual refactoring, we focused on the 66 Dockerfiles that were successfully built by both methods and analyzed improvements in build duration, image size, maintainability, and understandability for this intersection set.

Automated refactoring outperformed developer refactoring, reducing median image size twice as effectively As illustrated in Figure 6, the original Dockerfiles have a median image size of 599 MB. Developer refactoring reduces this to 340MB, a 43% reduction. The 50-shot LLM refactoring achieves a more substantial reduction, bringing the median size down to 95 MB, an 85% reduction.

Automated refactoring achieves a much greater reduction in build duration compared to developer refactoring.

As illustrated in Figure 6, the original Dockerfiles exhibit a median build duration of 58 seconds. The developer’s refactoring maintained the median build duration at 58 seconds, indicating that the central tendency of the build duration did not improve. The 50-shot LLM refactoring achieves a more significant reduction, bringing the median duration down to 46 seconds, a 19% reduction.

Automated refactoring shows a higher understandability improvement rate compared to developer refactorings For understandability, LLM refactoring improved 80% of cases (53/66), maintained 18% (12/66), and worsened one case. Developer improvement was 57% (38/66), maintained 38% (25/66), and worsened in three cases.

Automated refactoring shows a higher maintainability improvement rate compared to developer refactoring, with strong results for both. For maintainability, LLM refactoring improved 91% of cases (60/66), maintained 7% (5/66), and worsened one case. Developer improvement was 71% (47/66), maintained 21% (14/66), and worsened in five cases.

Finding-6. Automated refactoring outperformed developers’ manual refactorings on all metrics, reducing image size twice as much and achieving around 19-23% better improvements in build duration, understandability, and maintainability.

Developers and LLM exhibit different distributions of refactoring techniques, with the exception of the “Extract Stage” technique, which is frequently used by both. The plot in Figure 7 shows distinct patterns in refactoring techniques between developers and LLMs. Interestingly, both groups use the “Extract Stage” technique frequently, accounting for around 15% of total refactorings, highlighting a shared emphasis on this method. Additionally, developers predominantly use the “Inline Run Instruction” technique, representing 20% of their refactorings. In contrast, techniques like “Update Image Tag”, “Inline Stage”, and “Update Base Image” are less frequently used by developers, suggesting a lesser focus on these methods.

Case analysis: In Figure 7, we present a case to demonstrate the difference in Dockerfile refactoring strategies between the developer and the LLM. The original Dockerfile (pre-

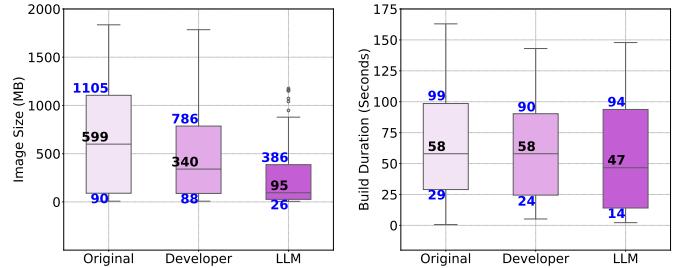


Figure 6: Docker image sizes and build durations across three categories: the original Dockerfiles (pre-refactoring), those refactored by developers, and those refactored using the 50-shot LLM approach.

refactoring), using a node:9.11 base image, resulted in an image size of 1110 MB and a build duration of 91 seconds. This Dockerfile included multiple steps such as copying dependency files, installing packages, building the application, and cleaning up unnecessary files. The human developer’s refactoring introduced significant improvements. The image size was reduced to 712 MB, and the build duration decreased to 73 seconds. This was achieved through “Inline Run Instruction” refactoring, which involved grouping commands in a single *RUN* instruction using `&&`. This technique minimizes the number of Docker layers created. Additionally, the developer removed the redundancy of two *COPY* commands, initially intended to optimize caching for dependency installation, by consolidating them into a single step. This further contributed to the reduced image size. In contrast, the LLM’s approach demonstrated a more complex strategy. The LLM employed a multi-stage build process, often referred to as “Extract Stage” refactoring. This approach ensured that only essential components for running the application were included in the final image, while all build dependencies were discarded. Furthermore, the LLM transitioned from a node:9.11 to a node:9.11-slim image for the final stage, a technique known as “Update Image Tag”. These actions significantly reduced the image size to 195 MB, although the build duration slightly increased to 94 seconds. The original Dockerfiles and those refactored by the developers and the language model all have similar levels of maintainability and understandability. This consistency is due to avoiding unspecified versions, which supports maintainability, and including detailed comments for each step, which enhances understandability.

Alongside the comparisons of automated and manual refactoring, we assessed the reduction in image size achieved through automated refactoring in contrast to the automated smell repair performed by PARFUM on the smelly Dockerfiles

Table II: Refactoring vs. Smell Fixing: Image Size Reduction and Build Success Rate

Tool / Approach	Build Success Rate (%)	Improvement Rate (%)	Deterioration Rate (%)	Average Reduction (MB)	Total Reduction (GB)
Smell-Repair (PARFUM[6])	50.4% (63/125)	69.8% (44/63)	30.2% (19/63)	12.2 MB (2.2%)	0.8 GB
Refactoring (ICL)	66.4% (83/125)	88.0% (73/83)	12.0% (10/83)	212.0 MB (25.6%)	17.1 GB

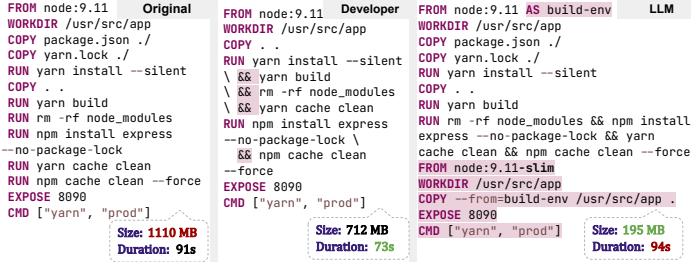


Figure 7: Illustrative Example of Dockerfile Refactoring: Manual vs. Automated (Commit 729ee76 from cars10/elasticvue; comments have been removed)

present in our dataset (125 Dockerfiles and 74 projects).

Automated refactoring achieves a 10x reduction in Dockerfile image size and a higher build success rate compared to automated smell repair. As illustrated in Table II, ICL refactoring achieved a build success rate of 66.4% (83/125), notably higher than PARFUM’s 50.4% (63/125). Moreover, automated refactoring yielded an improvement rate of 88.0%, surpassing PARFUM’s improvement rate of 69.8%. Notably, the average image size reduction achieved by ICL was 212.0 MB (25.6%), translating to a total reduction of 17.1 GB across all Dockerfiles. In comparison, PARFUM attained an average reduction of 12.2 MB (2.2%) and a total reduction of only 0.8 GB.

These findings reveal a key limitation of smell-focused approaches. While PARFUM offers modest image size gains and outperforms other smell-fixing tools [3], it remains confined to fixing specific issues within individual instructions, mainly addressing Bash commands in RUN instructions. By focusing narrowly on tasks like cache clearing, temporary file removal, and package installation cleanup (e.g., pip, npm, apt-get), PARFUM achieves limited image size reduction and fails to address structural inefficiencies in Dockerfiles. In contrast, refactoring adopts a design-level approach by reconfiguring all instructions and consolidating layers across stages.

D. RQ4: Build Failure Reasons

Despite notable improvements in various metrics, build failure rates remain relatively high for both manual and automated refactoring, at 53% (108/202) and 37% (74/202) respectively.

Ensuring Dockerfile refactorings result in functional builds is imperative; otherwise, the refactoring process defeats its purpose by causing build failures. A qualitative analysis of these failures reveals their primary causes.

Build context errors were the most frequent cause of failures in LLM-generated Dockerfiles, accounting for 52% (39/74) of the cases, compared to 33% (36/108) in developer-refactored Dockerfiles. Notably, there were 27 instances of build context errors common to both LLM-generated and developer-refactored Dockerfiles. These errors often arise from incorrect file paths or misconfigured build contexts. In this example [50], the error occurred because the path to “install.sh” in the *COPY* instruction did not consider the build context.

Dependency errors were the most prevalent issue in developer-refactored Dockerfiles, representing 43% (46/108) of failures, and were also significant in LLM-generated Dockerfiles at 27% (20/74), with 8 cases of overlap. These errors typically occur when software components, packages, libraries, or tools are missing, incompatible, or incorrectly configured. An example is found in [51], where developers performed an “Update Image Tag” to an older Debian “wheezy” base image, causing compatibility issues and “apt-get install” failures due to outdated repositories.

Syntax errors were more common in LLM-generated Dockerfiles, constituting 11% (8/74) of failures, compared to 6% (7/108) in developer-refactored ones, indicating a need for enhanced syntax validation in LLM processes. For instance, in one example, [52], the “**&&**” was missing when performing “Inline Run Instruction” refactoring.

Missing base images were more frequently an issue in developer-refactored Dockerfiles, occurring in 15% (16/108) of cases. This was often due to developers using local images not available on DockerHub, leading to build failures due to the inability to retrieve these private images. Conversely, the rate for LLM-generated Dockerfiles was 5% (4/74), primarily due to the generation of references to non-existent images. For example, a Dockerfile failed in [53] because developers performed an “Update Base Image” refactoring and specified a base image that was not present in Dockerhub.

Other errors, related to experimental settings such as resource limitations and network issues, were very less common.

V. THREATS TO VALIDITY

This study acknowledges several threats to validity. A potential one is the diversity of our dataset. Although we collected a large set of Dockerfiles using various refactoring keywords, but our results may not fully encompass Dockerfiles that require unique build steps. During data collection, we focused on immediate builds and excluded files needing additional setup or dependencies. However, the dataset’s inclusion of projects from various domains, with varying popularity and programming languages, provides a reasonably representative sample. Future research can replicate our study across different scenarios to ensure broader applicability. Additionally, there is a potential risk that some test Dockerfiles may have been encountered by GPT-4o during its pre-training phase, which could influence the results. However, the model’s suboptimal zero-shot performance and superior results compared to developer refactorings suggest that memorization is unlikely to be the primary driver of outcomes. The metrics used, such as image size and build duration, can be affected by factors beyond Dockerfile modification, such as application changes and Docker engine conditions. We mitigated this by standardizing testing conditions, including clearing Docker caches and using consistent commit points. We relied on DRMiner [28], a state-of-the-art tool, to detect refactorings between commits. Despite an F1 score of 0.94, DRMiner’s limitations may lead to missed or misidentified refactorings.

VI. CONCLUSION

Throughout the lifecycle of Docker projects, we observe that image size and build duration consistently increase. This phenomenon is often due to the gradual accumulation of dependencies and suboptimal configurations, with developers frequently postponing necessary refactorings.

In exploring the automation of refactoring, we demonstrated that LLMs can significantly streamline this process. Our experiments revealed that the effectiveness of these models improves with the number of refactoring demonstrations provided, with the 50-shot configuration yielding the most substantial gains. Specifically, this setup achieved superior results in reducing Docker image size and build duration and also improved maintainability and understandability compared to manual refactoring by developers. A key finding of our study is that refactoring involves trade-offs; enhancing one metric often negatively impacts others. Moreover, both automated and manual refactorings are susceptible to build failure, primarily due to context errors and dependency issues. This issue emphasizes the need for more robust error handling and validation mechanisms, potentially as part of future work.

Integrating automated refactoring into CI/CD pipelines can provide a more agile and responsive development environment, allowing for continuous improvement and optimization of Dockerfiles. This study contributes to the growing body of knowledge on the application of AI in IaC.

REFERENCES

- [1] R. Bullington-McGuire, A. K. Dennis, and M. Schwartz, *Docker for Developers: Develop and run your application with Docker containers using DevOps tools for continuous delivery*. Packt Publishing Ltd, 2020.
- [2] I. Miell and A. Sayers, *Docker in practice*. Simon and Schuster, 2019.
- [3] T. Durieux, “Empirical study of the docker smells impact on the image size,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–12.
- [4] Y. Wu, Y. Zhang, T. Wang, and H. Wang, “Characterizing the occurrence of dockerfile smells in open-source software: An empirical study,” *IEEE Access*, 2020.
- [5] Q.-C. Bui, M. Laukötter, and R. Scandariato, “Docker-cleaner: Automatic repair of security smells in dockerfiles,” in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2023, pp. 160–170.
- [6] T. Durieux, “Parfum: Detection and automatic repair of dockerfile smells,” *arXiv preprint arXiv:2302.01707*, 2023.
- [7] G. Rosa, S. Scalabrino, G. Robles, and R. Oliveto, “Not all dockerfile smells are the same: An empirical evaluation of hadolint writing practices by experts,” in *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 2024, pp. 231–241.
- [8] E. Ksontini, M. Kessentini, T. d. N. Ferreira, and F. Hassan, “Refactorings and technical debt for docker projects,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021.
- [9] M. Straesser, A. Bauer, R. Leppich, N. Herbst, K. Chard, I. Foster, and S. Kounev, “An empirical study of container image configurations and their impact on start times,” in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2023, pp. 94–105.
- [10] H. Azuma, S. Matsumoto, Y. Kamei, and S. Kusumoto, “An empirical study on self-admitted technical debt in dockerfiles,” *Empirical Software Engineering*, vol. 27, no. 2, p. 49, 2022.
- [11] Y. Wu, Y. Zhang, K. Xu, T. Wang, and H. Wang, “Understanding and predicting docker build duration: An empirical study of containerized workflow of oss projects,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.
- [12] Y. Wu, Y. Zhang, T. Wang, and H. Wang, “Dockerfile changes in practice: A large-scale empirical study of 4,110 projects on github,” in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*, 2020, pp. 247–256.
- [13] Y. Golubev, Z. Kurbatova, E. A. AlOmar, T. Bryksin, and M. W. Mkaouer, “One thousand and one stories: a large-scale survey of software refactoring,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1303–1313.
- [14] G. Rosa, A. Mastropaoletti, S. Scalabrino, G. Bavota, and R. Oliveto, “Automatically generating dockerfiles via deep learning: Challenges and promises,” in *2023 IEEE/ACM International Conference on Software and System Processes (ICSSP)*. IEEE Computer Society, 2023, pp. 1–12.
- [15] OpenAI, “Gpt-4 technical report,” <https://arxiv.org/pdf/2303.08774.pdf>, 2023, [Online; accessed 2024-11-08].
- [16] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, “Large language models for software engineering: A systematic literature review,” *ACM Transactions on Software Engineering and Methodology*, 2023.
- [17] N. Nashid, M. Sintaha, and A. Mesbah, “Retrieval-based prompt selection for code-related few-shot learning,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2450–2462.
- [18] O. Rubin, J. Herzig, and J. Berant, “Learning to retrieve prompts for in-context learning,” *arXiv preprint arXiv:2112.08633*, 2021.
- [19] D. Labs, “How to create dockerfiles with genai,” <https://www.docker.com/blog/how-to-create-dockerfiles-with-genai/>, 2024, [Online; accessed 2024-11-08].

- posted Jul. 29, 2024; accessed 2024-11-08].
- [20] E. Ksontini, M. Mastouri, R. Khalsi, and W. Kessentini, “Replication package msr 2025,” <https://sites.google.com/view/msrdra25/home>.
 - [21] Docker. (2024) Dockerfile best practices. Accessed: 2023-06-04. [Online]. Available: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
 - [22] T. Sharma, M. Fragkoulis, and D. Spinellis, “Does your configuration code smell?” in *Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR '16)*. IEEE, 2016, pp. 189–200.
 - [23] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps, “Learning from, understanding, and supporting devops artifacts for docker,” in *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*, 2020.
 - [24] iwei Xu, Y. Wu, Z. Lu, and T. Wang, “Dockerfile tf smell detection based on dynamic and static analysis methods,” in *Proceedings of the 43rd IEEE Annual Computer Software and Applications Conference (COMPSAC '19)*, 2019.
 - [25] Y. Wu, “Exploring the relationship between dockerfile quality and project characteristics,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 2020, pp. 128–130.
 - [26] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall, “An empirical analysis of the docker container ecosystem on github,” in *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories (MSR '17)*, 2017, pp. 323–333.
 - [27] J. Henkel, D. Silva, L. Teixeira, M. d’Amorim, and T. Reps, “Shipwright: A human-in-the-loop system for dockerfile repair,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1148–1160.
 - [28] E. Ksontini, A. Abid, R. Khalsi, and M. Kessentini, “Dr-miner: A tool for identifying and analyzing refactorings in dockerfile,” in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 584–594.
 - [29] G. Rosa, S. Scalabrino, and R. Oliveto, “Assessing and improving the quality of docker artifacts,” in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 592–596.
 - [30] A. M. Eilertsen, “Refactoring operations grounded in manual code changes,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 2020, pp. 182–185.
 - [31] M. Harman, S. A. Mansouri, and Y. Zhang, “Search-based software engineering: Trends, techniques and applications,” *ACM Computing Surveys*, vol. 45, no. 1, p. 11, 2012.
 - [32] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, “Search-based refactoring: Towards semantics preservation,” in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 347–356.
 - [33] V. Alizadeh, M. Kessentini, W. Mkaouer, M. Ocinneide, A. Ouni, and Y. Cai, “An interactive and dynamic search-based approach to software refactoring recommendations,” *IEEE Transactions on Software Engineering*, 2018.
 - [34] A. Nyamawe, H. Liu, N. Niu, Q. Umer, and Z. Niu, “Automated recommendation of software refactorings based on feature requests,” 09 2019, pp. 187–198.
 - [35] P. Sagar, E. Alomar, M. W. Mkaouer, A. Ouni, and C. Newman, “Comparing commit messages and source code metrics for the prediction refactoring activities,” *Algorithms*, vol. 14, p. 289, 09 2021.
 - [36] S. Gao, X.-C. Wen, C. Gao, W. Wang, H. Zhang, and M. R. Lyu, “What makes good in-context demonstrations for code intelligence tasks with llms?” *ASE*, 2023.
 - [37] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, and Z. Chen, “A critical review of large language model on software engineering: An example from chatgpt and automated program repair,” *arXiv preprint arXiv:2310.08879*, 2023.
 - [38] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao *et al.*, “Self-refine: Iterative refinement with self-feedback,” <https://arxiv.org/abs/XXXX.XXXXX%7D%7D>, 2023, arXiv preprint.
 - [39] A. Shirafuji, Y. Oda, J. Suzuki, M. Morishita, and Y. Watanobe, “Refactoring programs using large language models with few-shot examples,” in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2023, pp. 151–160.
 - [40] K. G. Srivatsa, S. Mukhopadhyay, G. Katrapati, and M. Shrivastava, “A survey of using large language models for generating infrastructure as code,” *arXiv preprint arXiv:2404.00227*, 2024.
 - [41] “hadolint/hadolint: Dockerfile linter, validate inline bash, written in haskell,” <https://github.com/hadolint/hadolint>, (Accessed on 11/17/2023).
 - [42] Google, “Bigquery github database,” <https://codelabs.developers.google.com/codelabs/bigquery-github>.
 - [43] E. AI Omar, M. W. Mkaouer, and A. Ouni, “Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages,” in *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*. IEEE, 2019, pp. 51–58.
 - [44] S. Min, X. Lyu, A. Holtzman, M. Artetxe, M. Lewis, H. Hajishirzi, and L. Zettlemoyer, “Rethinking the role of demonstrations: What makes in-context learning work?” *arXiv preprint arXiv:2202.12837*, 2022.
 - [45] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, M. Gatford *et al.*, “Okapi at trec-3,” *Nist Special Publication Sp*, vol. 109, p. 109, 1995.
 - [46] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*,

- vol. 33, pp. 1877–1901, 2020.
- [47] R. A. Poldrack, T. Lu, and G. Beguš, “Ai-assisted coding: Experiments with gpt-4,” *arXiv preprint arXiv:2304.13187*, 2023.
 - [48] Z. Namrud, K. Sarda, M. Litoiu, L. Shwartz, and I. Watts, “Kubeplaybook: A repository of ansible playbooks for kubernetes auto-remediation with llms,” in *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, 2024, pp. 57–61.
 - [49] C. Spearman, “The proof and measurement of association between two things.” 1961.
 - [50] “Build context error example, project: zhoumingithub/my-zipkin, commit:4e74b2, dockerfile: docker/dockerfile,” <https://github.com/zhoumingithub/my-zipkin/commit/4e74b2>.
 - [51] “Dependency errors example, project: mattolson/docker-base, commit:ee4d42, dockerfile: Dockerfile,” <https://github.com/mattolson/docker-base/commit/ee4d42>.
 - [52] “Syntax errors example, project: ipcjk/ixgen, commit:9e1f90, dockerfile: docker/dockerfile,” <https://github.com/ipcjk/ixgen/commit/9e1f90>.
 - [53] “Missing base images example, project: ekumenlabs/terminus, commit:e2305d, dockerfile: docker/gazebo-terminus-intel-dev/dockerfile,” <https://github.com/ekumenlabs/terminus/commit/e2305d>.