DETECTING AND REFACTORING TECHNICAL DEBT FOR SOFTWARE CONTAINERS

by

EMNA KSONTINI

A dissertation submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND INFORMATICS

2024

Oakland University
Rochester, Michigan

Doctoral Advisory Committee:

Hua Ming, Ph.D., Chair
Marouane Kessentini, Ph.D.
Steven Wilson, Ph.D.
Mehdi Bagherzadeh, Ph.D.
Sam Srauy, Ph.D.

As a token of respect, gratitude, and recognition, I dedicate this work:

To God the most merciful.

To my source of life, happiness, joy, and motivation, the strongest and most tender woman I have ever known, my dear lovely mother.

To my source of courage, stability, and motivation, the man of sacrifices, my dear father.

To my rock and confidant, my dear partner, your unwavering support and love have been my guiding light. Thank you for existing in this journey.

To my adorable aunties, you have always been there for me. Thank you for your endless warmth, love, and companionship.

To my professors, for their advice, patience, and faith. I am very grateful.

To my closest friends, my second family, for their unconditional support.

To anyone from near or far who has helped me cross a horizon in my life.

Thank you.

—

ACKNOWLEDGMENTS

At the end of this work, I would like to express my deep gratitude to all those who supported me and helped me accomplish it in the best conditions.

My profound gratitude goes first to my supervisor, Prof. Marouane Kessentini, who expertly guided me through this graduation project. His unwavering enthusiasm for the search-based software engineering domain kept me constantly engaged with my research. His guidance, persistent help, insightful feedback, and scientific approaches have greatly helped me accomplish this work.

Next, I would like to thank the Department of Computer Science and Engineering at Oakland University for providing the required support that allowed me to complete my thesis.

I want to thank the honorable committee members for agreeing to examine this report and for taking the time to analyze and attend the academic defense of this little effort. I'm hopeful they'll discover the clarity and purposefulness they seek.

My appreciation goes out to my partner, Amine Barrak, for his tremendous understanding and encouragement over the past few years. Your support has meant more to me than you could realize.

I am also grateful to my current and former ISE-LAB members for their kindness and favorable and friendly atmosphere, which made my time working with the lab enjoyable.

Finally, I humbly thank all concerned parties who co-operated with me in this work.

ABSTRACT

DETECTING AND REFACTORING TECHNICAL DEBT FOR SOFTWARE CONTAINERS

by

EMNA KSONTINI

Committee chair: Hua Ming, Ph.D.

In today's fast-paced software development environment, containerization has emerged as a cornerstone of modern infrastructure, enabling consistent and scalable deployments across varied platforms. Docker, as the leading containerization platform, plays a critical role in this landscape, but the management of Dockerfiles—scripts that automate the creation of container images—presents significant challenges. These challenges include the accumulation of technical debt, inefficient image sizes, prolonged build durations, and the emergence of anti-patterns, all of which can undermine the efficiency, maintainability, and quality of Docker-based projects.

This dissertation presents and advances a suite of techniques and tools to enhance the quality of Docker projects through carefully targeted refactoring strategies, anti-pattern detection, and the mitigation of technical debt. It begins with an empirical study of open-source Docker projects, identifying 38 Docker-specific refactoring techniques and nine distinct categories of technical debt. These findings illuminate the unique challenges inherent in Dockerfile management, which differ fundamentally from those in traditional software development due to Docker's Infrastructure as Code (IaC) nature.

Building on these insights, the research introduces DRMiner, a tool designed to detect and analyze refactorings within Dockerfiles. Utilizing an Enhanced Abstract Syntax Tree (E-AST) approach, DRMiner addresses the specific complexities of Docker artifacts,

automating the identification of refactoring opportunities and significantly reducing the manual effort required for Dockerfile maintenance.

The dissertation also proposes a novel method for the specification and detection of Docker-specific anti-patterns, expanding the scope beyond traditional code smells to address broader design flaws. This method defines five new anti-patterns and develops a metric-based framework for their automated detection,

Finally, this research delves into automating Dockerfile refactoring through large language models (LLMs). The findings reveal that LLM-driven refactoring reduces image sizes and build durations and enhances maintainability and understandability, outperforming manual refactoring methods.

Together, these contributions provide a robust, empirically grounded framework for enhancing the quality and sustainability of Docker projects, offering valuable tools and insights for both practitioners and researchers in containerization.

TABLE OF CONTENTS

TABLE OF CONTENTS—Continued

TABLE OF CONTENTS—Continued

LIST OF FIGURES

LIST OF TABLES

CHAPTER ONE

INTRODUCTION

1.1 Research Context

In the rapidly evolving and highly competitive field of technology, IT companies are continually challenged to remain agile and responsive to the swift pace of digital innovation. This dynamic environment necessitates the rapid development and deployment of new features and services, compelling organizations to adopt more efficient and scalable software development methodologies [1]. One methodology that has gained significant traction is containerization, with Docker serving as a prime example. The "Annual Container Adoption" report in 2020 [1] found that 79% of companies chose Docker as their main container technology. Containerization has become a crucial strategy for enhancing productivity and ensuring consistency across development, testing, and production environments [2].

Containerization involves packaging an application along with all its dependencies—such as libraries, configuration files, and binaries—into a single, self-contained unit known as a container. These containers can run consistently across various computing environments, from a developer's local machine to testing and production servers. The primary advantage of this approach is that it isolates the application from the underlying infrastructure, ensuring consistent behavior regardless of deployment location. This isolation simplifies dependency management and streamlines deployment processes, allowing developers to focus on crafting solutions without worrying about the complexities of different runtime environments and compatibility issues. Containers are lightweight and modular, promoting an efficient, scalable, and secure software development lifecycle.

However, the flexibility and agility that containerization offers also introduce certain challenges. As software evolves, frequent updates to Docker configurations and container images are often required. This can lead to the risk of adopting inefficient practices or "anti-patterns". While these practices might provide immediate benefits, they can introduce complexity and contribute to the accumulation of technical debt over time. Technical debt occurs when short-term solutions are chosen over more robust and sustainable ones, leading to future costs and complications. This debt can ultimately make the system more difficult and expensive to maintain.

Moreover, the relentless pressure to innovate often leads developers to prioritize feature development and code enhancements over maintaining the quality of their Docker environme -nts. This oversight can result in the proliferation of issues within containers, undermining their potential for reusability, lightweight deployment, and security [3, 4, 5, 1].

1.2 Problem Statement

Docker, the industry standard for containerization, leverages Infrastructure as Code (IaC) principles to facilitate the creation of images through Dockerfiles. 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 [6].

Despite being the backbone of containerization, Dockerfiles are often plagued with problems such as smells [7]. In this context, several works have been proposed to detect and fix them [5, 8, 9]. 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 [5, 10]. While bash-based solutions are useful, they fall short of 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**, 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 [11, 7]. **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 [12]. **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 [13]. As a project progresses through its development stages, the content of the Dockerfile may be revised many times [14], which can affect **understandability** and complicate subsequent updates.

Refactoring, defined as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior" [15], is commonly used in industry to reduce technical debts. Many refactoring tools are designed for languages like Java, JavaScript, C, and Python [16, 17, 18, 19]. However, they are inadequate for Dockerfiles due to the distinct nature of Infrastructure as Code and their declarative syntax. Dockerfiles are used to containerize applications across various programming languages (e.g., Java, PHP, Go) and domains (e.g., machine learning (ML), web development). However, these tools are not suitable for Docker projects. This is due to Docker artifacts' unique IaC nature and syntax, which significantly differ from traditional source code, necessitating specialized research and tools. Adding to the above introduced

challenges, the refactoring process for Dockerfiles has not been formally defined. Unlike traditional programming languages where refactoring techniques and best practices have been extensively studied and documented. This absence of formal guidelines and tool means that developers must rely on their own experience and ad-hoc methods to identify problems, improve, and maintain their Dockerfiles. This manual optimization of Dockerfiles is both time-consuming and error-prone, requiring significant expertise in Docker and the underlying infrastructure.

1.3 Proposed Research

Our proposed research, illustrated in Figure 1.1, is structured into two key phases. The foundational phase (Phase 1) involves leveraging extensive data from GitHub, a leading open-source code repository, to extract patterns from the commit histories of Docker projects. By analyzing commit descriptions and project-specific details, we aim to correlate developer practices on Docker artifacts with subsequent changes in overall project quality. This analysis will enable us to define categories of Docker-specific technical debt, refactoring techniques, and common anti-patterns.

Building on these empirically derived insights, Phase 2 focuses on developing a comprehensive framework designed to enhance the quality of Docker projects. This framework will integrate tools capable of detecting anti-patterns and identifying quality issues within Docker artifacts. Additionally, it will provide automated refactoring recommendations,

The proposed research serves a dual purpose. For practitioners and educators, it offers a systematic approach to improving Docker project quality by encapsulating best practices and promoting a culture of continuous improvement. For researchers, it establishes a foundational platform for further investigation into Docker project management and quality evolution. By addressing current deficiencies and providing structured guidance, this framework aspires to significantly advance the state of containerization technology.

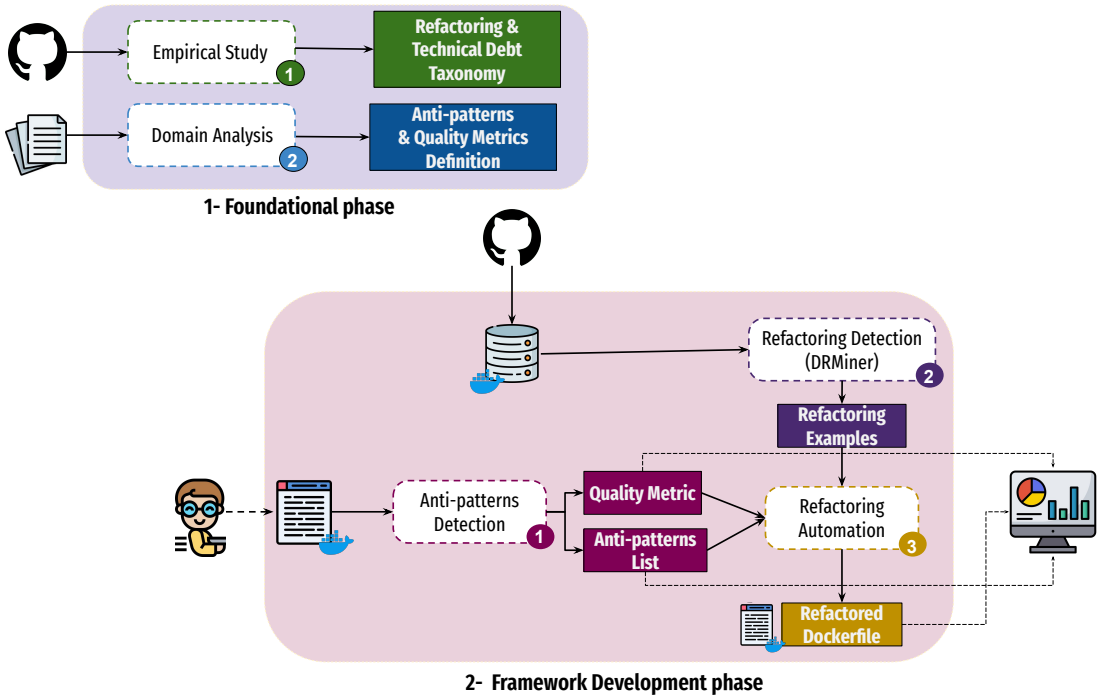


Figure 1.1: Proposed Research Overview

In this context, we were able to publish and submit four contributions, all in top-tier journals and conferences.

1.4 Research Contributions

In the following, we will summarize the objectives of each contribution. Unlike regular source code, Docker artifacts do not have a defined set of refactoring techniques or a clear understanding of technical debt. To fill this gap and systematically identify and categorize these aspects, we initiate our dissertation with a comprehensive empirical study.

Contribution 1: A Taxonomy of Docker-Specific Refactoring and Technical debts: Like any other complex system, container-based projects are prone to various quality and technical debt issues related to their artifacts: Docker and Docker-compose files. Unfortunately, there is a knowledge gap in how container-based projects evolve and

are maintained. This PhD work addresses the above gap by studying refactorings, i.e., structural changes while preserving the behavior, applied in open-source Docker projects, and the technical debt issues they alleviate. We analyzed 68 projects, including 19,5 MLOC and 193 manually examined commits to introduce new Docker -specific refactorings and technical debt categories and define different best practices.

Although the importance of refactoring inside Docker ecosystems is apparent, detecting it remains challenging. Developers usually avoid documenting their refactoring efforts, often combining them with other changes. For this reason, we propose a tool tailored for the identification and analysis of Dockerfile-specific refactoring

Contribution 2: DRMiner: A Tool For Identifying And Analyzing Refactorings In Dockerfile While previous contributions have delved into defining Docker refactoring, the predominant focus has been on empirical foundations. There remains a clear gap for an exhaustive tool that can adeptly navigate the complexities of Dockerfile refactoring detection. To fill this gap, we introduce DRMiner, the first tool for identifying and analyzing refactoring in Dockerfile. Our solution, relies on a novel E-AST (Enhanced Abstract Syntax Tree) based component-matching algorithm and a set of detection rules to determine refactoring candidates.

Refactorings are frequently carried out to fix anti-patterns, which are common coding practices that, while not incorrect, contribute to poor design and can compromise system quality over the long term. However, there is currently a dearth of studies specifically dedicated to defining and identifying anti-patterns within Docker artifacts. In response to this gap, we propose a method and tool for specifying and detecting anti-patterns within Docker artifacts.

Contribution 3: A Method for Specification and Detection of Dockerfile Anti-patterns Existing research has focused on identifying Dockerfile "smells,"

highlighting coding and bash script issues. However, these fail to address broader design flaws. Dockerfile anti-patterns encompass a broader scope, affecting the overall design and architecture of Docker images and potentially leading to more significant issues such as reduced maintainability, increased technical debt, and decreased system performance. Recognizing this gap, we introduce a novel approach for specifying and automatically detecting Dockerfile anti-patterns. The process starts with an extensive domain analysis to define new Dockerfile anti-patterns commonly found in the industry. These anti-patterns are then formalized using a set of metrics that allow for their detection through static analysis.

The detection of anti-patterns and the insights gained from developer refactoring practices using tools like DRMiner are crucial for maintaining Dockerfile quality. However, the manual effort required to address these issues and implement those practices often leads to significant delays and inefficiencies. To mitigate these challenges, we explore the automation of refactoring, with a focus on enhancing both the speed and reliability of the process.

Contribution 4: Dockerfile Refactoring Automation via In-Context Learning Maintaining Dockerfiles presents significant challenges. Refactoring, in particular, is often a manual and complex process. We examine the feasibility and practicality of automating Dockerfile refactoring across 600 Dockerfiles from 358 open-source projects. We introduce a novel methodology that leverages large language models (LLMs) and in-context learning to automate this task. Our approach begins with a thorough analysis of current practices in Dockerfile refactoring, providing critical insights into the potential advantages of automation. We then assess the effectiveness of LLMs in automating refactoring, utilizing varied prompting strategies and a novel score-based selection method to enhance metrics such as image size, build duration,

understandability, and maintainability. Furthermore, this work includes a comparative analysis of LLM-driven automation versus traditional manual refactoring, emphasizing the efficiency and benefits of automated approaches. Finally, we investigate the root causes of build failures associated with automated and manual refactorings.

1.5 Publications List

- **Emna Ksontini**, Marouane Kessentini, Thiago do N. Ferreira and Foyzul Hassan "**Refactorings and Technical Debt in Docker Projects: An Empirical Study**." published in the 36th IEEE/ACM International Conference on Automated Software Engineering ASE (2021). DOI 10.1109/ASE51524.2021.9678585. **Acceptance rate 16%**.

- **Emna Ksontini**, Rania Khalsi and Marouane Kessentini (2024) "**A Taxonomy on Docker-specific Refactoring and Technical Debts**", under review in IEEE Transactions on Software Engineering (TSE). **Impact Factor 6.11**.

- **Emna Ksontini**, Aycha abid, Rania Khalsi and Marouane Kessentini"**DRMiner: A Tool For Identifying And Analyzing Refactorings In Dockerfile**." published in the 21 st International Conference On Mining Software Repositories MSR (2024). DOI 10.1145/3643991.364492. **Acceptance rate 26.3%**.

- **Emna Ksontini**, Rania Khalsi and Marouane Kessentini (2024) "**Specification and Detection of Docker-specific Anti-patterns**", under review in IEEE Transactions on Software Engineering (TSE). **Impact Factor 6.11**.

- **Emna Ksontini**, Meriem Mastouri, Rania Khalsi and Marouane Kessentini "**Dockerfile Refactoring Automation via In-context Learning**", submitted to the IEEE/ACM International Conference on Software Engineering ICSE (2025). **Acceptance rate 15%**.

1.6 Organization of the Dissertation

This thesis is organized as follows: Chapter II provides the necessary background information to understand the research contributions. Chapter III offers a summary of the related work. Chapter VI presents an empirical study that leads to the creation of a taxonomy for Docker-specific refactoring practices and technical debt. Chapter V describes the approach and tool developed to detect Docker-specific refactoring practices. Chapter VI introduces a method for identifying existing anti-patterns in open-source projects and a tool designed to detect them. Chapter VII outlines the approach used to automate Dockerfile refactoring through in-context learning. Finally, Chapter VII concludes with a summary of the research and discusses potential future directions.

CHAPTER TWO

BACKGROUND

This chapter provides the essential background knowledge required to fully understand the research presented in this thesis. We first explore fundamental concepts such as Docker and container-based projects, which form the core of modern application development. Next, we delve into the concepts of software refactoring, anti-patterns, and technical debt—key aspects that influence the quality and maintainability of software systems. By establishing a clear understanding of these topics, this chapter lays the foundation for the methodologies and analyses discussed in subsequent chapters of this thesis.

2.1 Docker and Container-based Projects

2.1.1 What is Containerization ?

Modern application development aims to keep apps running within the same host, isolated from one another so that they don't affect each other's operation or maintenance. However, applications may be written in a variety of languages, they may depend on third-party software or may be installed from an online repository, and generally have certain system requirements (such as memory size) to execute successfully. Due to all these constraints, isolation may be difficult.

Virtual machines can be considered as one solution to guarantee isolation [20], as they keep applications on the same hardware completely isolated, reducing conflicts between software components and competition for hardware resources to a minimum. However, virtual machines are difficult to maintain and update since they are pretty large (each virtual machine has its own operating system, which is often terabytes in size).

Containerization [20], on the other hand, is a mechanism that can be used to tackle this issue efficiently. Containerization enables packaging software requirements, including

features like binaries, libraries, configuration files, and dependencies, to be integrated into a unique and granular run-time environment (container) so that programs can execute reliably and evenly on any infrastructure. To put it another way, "containers allow application developers to construct their own kitchenette on servers wherever they go, rather than cooking in someone else's kitchen" [21].

Unlike virtual machines, containers are isolated in the sense that they don't include a copy of an operating system. Instead, a run-time engine (such as Docker engine) is installed on the host's operating system, serving as a duct for containers to share an operating system with other containers on the same machine. Many containers may be supported by a single host and can spin up and down in mass with far less effort and overhead.

2.1.2 Docker Architecture and Components

Docker[22] is an open-source platform for building, delivering, and executing applications. Docker may also be described as a container-management service. Docker's major goal is to make it simple to create applications and then distribute them into containers so that they can be deployed anywhere. Since its first release in March 2013, Docker has been a catchphrase for contemporary global development, particularly in the context of Agile-based projects. In the following paragraphs, we will be introducing Docker's main components, then we will give an overview of its architecture. Finally, we will detail Dockers artifacts that are the subject of this study.

2.1.2.1 Docker Image A Docker image is a read-only, executable blueprint that is being used to generate containers. The image includes all of the code, libraries, configuration files, and other dependencies required to run an application uniformly and consistently on any infrastructure.

2.1.2.2 Docker Registry A Docker registry is a storage that holds and distributes named Docker images in different tagged versions. The most popular container registry is DockerHub [23], which is the standard public registry for Docker.

2.1.2.3 Docker Architecture Docker architecture as illustrated in Figure 2.1, follows a client-server model and comprises three main components: Docker client, Docker Host, and Registry/HUB. The Docker client offers a command line interface that enables users to interact with the Docker server or daemon, which in turn does all the work of creating, executing, and terminating applications. When the Docker daemon receives a request from a client to create a container, it fetches the requested image from the Docker registry or the local image registry and creates a runnable image instance also named container.

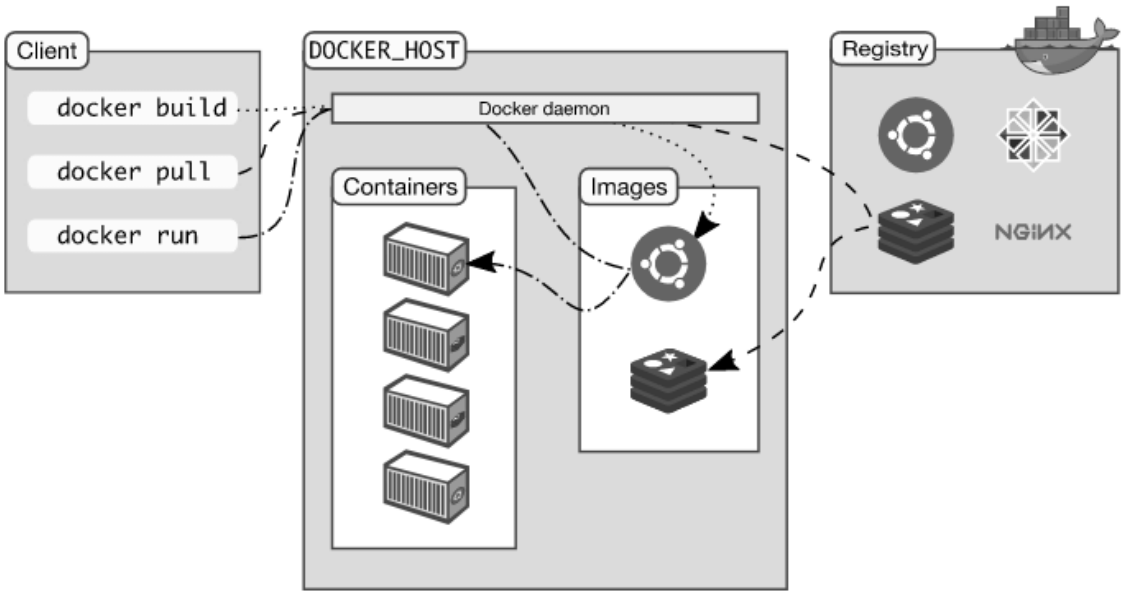


Figure 2.1: Docker Architecture [24]

2.1.2.4 Dockerfile Docker container images are built using `Dockerfile`, a document containing a sequence of instructions used for creating the computational environment, following the notion of Infrastructure-as-Code (IaC) [25].

```
1  FROM node:argon
2  # Create app directory
3  WORKDIR /usr/src/app
4  # Install app dependencies
5  COPY package*.json /usr/src/app/
6  RUN npm install
7  # Bundle app source
8  COPY . /usr/src/app
9  # Expose the app to the outside world
10 EXPOSE 8080
11 CMD [ "npm", "start" ]
```

Listing 1: Dockerfile example

An example of `Dockerfile` is illustrated in Listing 1. It starts from a previously existing base image defined by the FROM instruction, which acts as a starting point from which the Docker image will inherit infrastructure definitions. This parent image can be an official Docker image (e.g., alpine) or any other existing image (e.g., with pre-installed software). In order to suit the application needs and to create the desired environment, Dockerfile offers a list of setup instructions, which can be listed in Table 2.1.

Docker implements Union File Systems [4] when building images, adding filesystem layer upon layer. Docker puts a new layer to the stack for each instruction of the Dockerfile. Each of these layers is read-only. When the Docker daemon starts a container, it creates a writable layer on top of these layers. As illustrated in Figure 2.2, the Dockerfile contains seven instructions, each of which creates a layer with a specific size.

Table 2.1: Dockerfile setup instructions

Instruction	Description
ENV	Setting the environment variables
ARG	Defining variables that can be set at build time
WORKDIR	Setting working directory for all subsequent instructions
COPY	Copying files from host to the Docker image
ADD	Similar to COPY instruction but supports two additional tricks. It supports the use of URL instead of a local file and can recognize the archive format and extract it directly into the destination
LABEL	Key value pairs, indicating image metadata
RUN	Executing any command
EXPOSE	Informs Docker that the container is exposing a particular port
CMD	Setting a command and/or parameters that executes when the container is starting and which can be overwritten at build time
ENTRYPOINT	Setting executable that will always run when the container is initiated and cannot be overwritten.

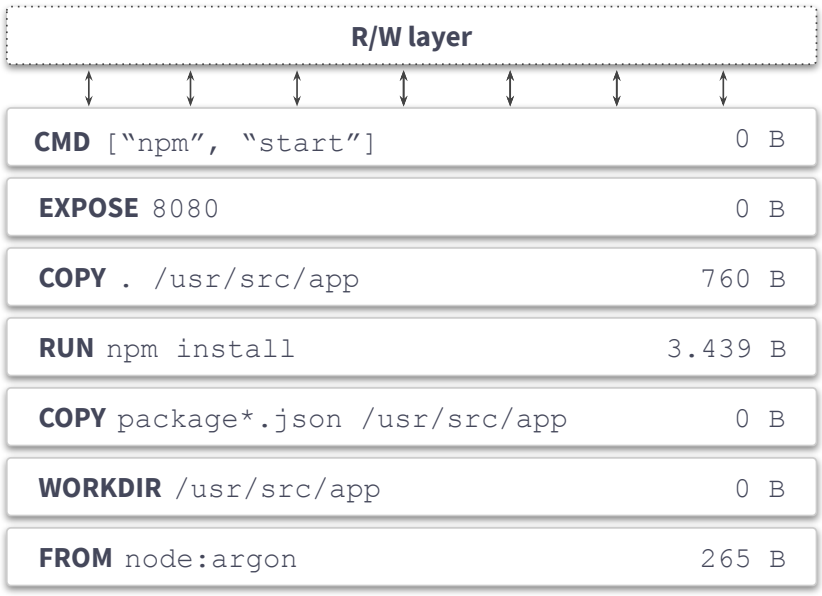


Figure 2.2: Dockerfile Layers

2.1.2.5 Docker-Compose In the Docker paradigm, each container captures one particular component of the software (e.g., database). Thus, when creating a multi-component application using Docker, it is unavoidable to combine multiple software components (containers) into an intricate workflow. To tackle this challenge, containers need to be instantiated and properly integrated. Docker-compose [26], can be used to mitigate this challenge by providing a unified setup routine that deploys several containers using a YAML configuration file, as known as `Docker-compose`.

```
version: "3.7"
    services:
        server:
            build: .
            ports:
                - 8080:4040
            environment:
                - DB_ADDRESS=database-mongo
                - DB_PORT=27017
                - PORT=4040
            depends_on:
                - database
        database:
            image: mongo:latest
            volumes:
                - mydata:/data/db
    volumes:
        mydata:
```

Listing 2: Docker-compose example

An example of a Docker-compose file is available in Listing 2. The example shows that the Docker-compose file is composed of two components/containers (SERVER and DATABASE). The SERVER component is represented by a local image (built from a `Dockerfile`, for instance that one available in Listing 1) and the DATABASE component is created from the "mongo" image, hosted in DockerHub [23] (an online registry for Docker Images). The docker-compose file also provides a list of setup attributes which can be listed in Table 2.2.

Table 2.2: Docker-compose setup attributes

Attribute	Description
BUILD	Setting path to the build context
IMAGE	Setting the image to start the container from
PORTS	Specify ports binding
ENVIRONMENT	Setting environment variables
DEPENDS_ON	Expressing dependency between services
VOLUMES	Setting volume bindings (host paths or named volumes)

Any typical Docker project includes the above files along with source code files written in typical programming languages, such as Java, to host the containers and enable their executions and synchronization with other features of the app that may not be containerized.

2.2 Refactoring, Anti-patterns and Technical Debt

In software development, the concept of **technical debt** is used to describe the implied cost of additional rework that arises when a simpler, quicker solution is chosen over a more thorough and time-consuming approach during the design and evolution of software systems [27]. Technical debt accumulates when developers prioritize immediate results over long-term code quality, leading to issues that will eventually need to be addressed, often at a higher cost.

Anti-patterns are a specific type of technical debt. They represent recurring solutions to common problems that may seem beneficial at first but ultimately lead to significant issues, exacerbating the technical debt. Unlike best practices, which guide developers toward optimal solutions, anti-patterns highlight common mistakes that should be avoided to improve project outcomes [28]. By encouraging solutions that are easier to implement in the short term but problematic in the long run, anti-patterns contribute directly to the accumulation of technical debt.

To manage and mitigate technical debt, especially that which arises from anti-patterns, **refactoring** is a widely used practice in the software development industry. Martin Fowler [15] defines refactoring as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior." This definition emphasizes that refactoring is a strategic approach to improving the internal quality of code. By reorganizing and optimizing the code structure, refactoring enhances maintainability, extensibility, and reusability, all while preserving the software's external functionality.

In summary, technical debt represents the future cost of expedient but suboptimal decisions made during software development. Anti-patterns are specific examples of such decisions that increase technical debt by promoting flawed solutions. Refactoring, in turn, is the process of systematically addressing these flaws, thereby reducing technical debt and improving overall code quality. Understanding the interplay between these concepts is crucial for developing and maintaining high-quality software systems.

CHAPTER THREE

STATE OF THE ART

3.0 Introduction

This chapter describes the related works that influenced and informed the development of this thesis. We begin by delving into the specific quality issues inherent in Dockerfiles, with a focus on works related to Dockerfile smells. Next, we review a set of empirical studies on software refactoring, shedding light on the strategies they employ to identify and define refactoring practices. Following this, we delve into repository mining techniques as a means of identifying refactorings and then discuss various refactoring detection tools designed to automate this process. We then turn our attention to the specification and detection of anti-patterns, discussing the methodologies that have been developed to address these pervasive issues in software systems. Finally, we explore the the field of refactoring automation.

3.1 Docker Quality Issues

Docker's documentation offers best practices [29] for addressing Dockerfile smells. However, developers' adherence to these guidelines is inconsistent. These smells, akin to traditional configuration code smells [30], indicate design flaws that, although not bugs, negatively affect Docker images.

In [5], 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. [7] have investigated how shell smells in Dockerfiles increase image size, affecting distribution efficiency and scalability.

Henkel et al. [31] have developed Binnacle to mine Bash-related smells from over 178,000 Dockerfiles. Xu et al. [32] 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* [9] detects and automatically repairs Dockerfile smells, evaluating repair effectiveness in terms of build failure and image size. Rosa et al.[10] have evaluated Hadolint writing practices by experts and ranked 26 additional Dockerfile smells, offering guidance for high-quality Dockerfile writing.

Other studies, such as [1], used linear regression to analyze Dockerfile quality and project characteristics, revealing statistical correlations. Cito et al. [2] found that missing version pinning smell in Dockerfiles often causes build issues, leading to non-reproducible builds. In [33], 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 improving performance and efficiency in

Despite the progress made, the current body of work largely focuses on detecting and fixing issues within the shell script embedded in the RUN instruction. There remains a gap in research dedicated to suggesting refactorings, identifying technical debt categories, and addressing anti-patterns—including Docker-specific smells—in Docker Compose.

3.2 Empirical Studies on Software Refactoring

Empirical studies on software refactoring investigated the way how developers are refactoring their code. Murphy-Hill et al. [34] investigated how developers perform refactorings. Examples of the exploited datasets are usage data from 41 developers using the Eclipse environment and information extracted from versioning systems. Among their several findings, they show that developers often perform floss refactoring, namely, they interleave refactoring with other programming activities, confirming that refactoring is rarely performed in isolation. Kim et al. [35] present a survey of software refactoring with 328 Microsoft engineers to investigate when and how they refactor code and developers' perceptions of the benefits, risks, and challenges of refactoring. They show that the major risk factor perceived by developers with regards to refactoring is the introduction of bugs and one of the main benefits they expect is to have fewer bugs in the future, thus indicating the usefulness of refactoring for code components exhibiting high fault-proneness. On top of that, 46% of the developers said that they mostly refactor during bug fixes and new feature implementation. Other works on empirical studies for refactoring analyzed the relationship between refactoring and bugs [36, 37, 38], showing that refactoring activities can be related to introducing bugs.

To the best of our knowledge, this thesis represents the first in-depth study dedicated to refactoring within Docker projects. While previous research has predominantly focused on traditional software paradigms, these approaches are not readily applicable to Docker artifacts due to their distinct Infrastructure as Code (IaC) characteristics.

3.3 Repository Mining Technique for Refactoring Identification

Many studies opted for repository mining as a way to identify refactoring activities. Weibgerber and Diehl [39] established the first approach for identifying local-scope and

class-level refactorings in CVS[1] repositories commit history. Their method consisted mainly of comparing the bodies of code components that are candidates for refactorings. Dig et al. [40] presented a tool named Refactoring Crawler that is able to detect refactoring between two versions of a component based on a combination of syntactic and semantic analysis.

Moreover, the appeal of GitHub is quickly growing, and it is commonly utilized as the foundation for data collection in the literature. Valerio et al. [41] claimed in a study conducted in 2016 that 73% of the empirical methods applied in literature relied on the direct inspection of Github meta-data and that in 39.8% and 12.8% percent of publications, respectively, used the Github API and Github Search API to collect data. Tsantalis et al. [42] developed RefactoringMine a lightweight and UMLDiff based algorithm that mines refactorings within Git commits history. Yiming et al. [43] studied refactoring applied in Machine Learning component by manually analysing GitHub projects and examining code patches changes within commits history.

3.4 Refactoring Detection Tools

In 2000, Demeyer et al. [44] introduced a reverse engineering technique that can interpret restructured code. This research enabled the development of sophisticated software refactoring detection tools. To detect clones, *CCfinder* [19] uses input source text transformation and token-by-token comparison. Antoniol et al. [45] developed a method to detect class-level refactorings across software releases using Vector Space Cosine Similarity of class identifiers. In addition, Xing and Stroulia [46] created the *UMLDiff* algorithm to assess changes in object-oriented design elements like packages, classes, and methods.

Some researchers used the above algorithms and ideas to create complete refactoring detection systems that automatically detect various code refactorings. The earliest detection

[1] CVS is an old centralized version control system similar to Github

tool developed is *RefactoringCrawler* by Dig et al. [18]. This tool was implemented as a plugin for Eclipse and can detect seven types of refactorings focusing on *rename* and *move* refactoring in Java code. In 2010, Kyle Prete's et al. *Ref-Finder* [47], leveraging *LSdiff* (Logical Structural Diff) [48] to analyze source code changes. Later, in 2017, Danilo et al. [16] introduced *RefDiff*, an automated tool identifying 13 refactoring types in git repositories by using a similarity index. In 2018, Tsantalis et al. [17] launched *REFACTORINGMINER*, a pioneering refactoring mining tool that identifies 15 refactoring types in git repositories without requiring code similarity thresholds. Using top-down matching of code entities, it achieves a precision of 98% and a recall of 87%.

This progress in refactoring detection technology has sparked diverse research studies. Stroggylos and Spinellis [49] used **such tools** to show that refactoring can deteriorate specific project code metrics. Another study [50] used *RefDiff* to reveal that 6.5% of lines deemed bug-introducing were actual refactorings. A different investigation [51], employing *RefDiff*, detected several type changes in Java projects, suggesting that type changes are more frequent than renaming.

While refactoring detection tools have greatly enhanced our understanding of code quality and evolution in software projects, their focus has largely been on source code. This leaves a notable gap in the realm of IaC, particularly Dockerfiles, which are essential for defining application environments.

3.5 Anti-patterns Specification and Detection

The negative impact of anti-patterns on software quality highlighted by number of empirical studies [52, 53, 54] has motivated the development of various automatic detection approaches. Most of these approaches attempt to identify bad motifs in models of source code.

Initial strides in anti-pattern detection were marked by the development of metric-based methodologies. Marinescu's seminal work [55] introduced "detection strategies" that leverage metrics to capture violations of good design principles in source code. Building on this, Moha et al. [56] proposed the DECOR method, utilizing a Domain-Specific Language (DSL) to specify source code anti-patterns through high-level abstractions. This method represents a significant advance by simplifying the specification and detection of complex anti-patterns, facilitating broader and more effective identification efforts.

Recent developments have seen the integration of machine learning techniques into anti-pattern detection. Afrin et al. [57] showcased a hybrid model that combines Artificial Neural Networks (ANN) and Support Vector Machines (SVM) to outperform previous methods in accuracy, precision, and recall.

The scope of anti-pattern detection has also expanded to include different domains beside source code. El-Dahshan et al. [53] delved into the domain of mobile applications, underscoring how anti-patterns can significantly diminish the quality and performance of these apps. Their study highlighted that mobile applications plagued by anti-patterns tend to suffer from degraded quality, resulting in a shorter lifespan. Saluja and Batra [58] explored the realm of web services, where the quality of service (QoS) parameters such as delay, throughput, reliability, and cost are of paramount importance. Their analysis, which focused on web service description files based on static metrics, provided insights into the early identification of anti-patterns, enabling timely interventions to mitigate their adverse effects on service quality.

Overall, the related work in anti-patterns specification and detection has demonstrated the utility of anti-patterns in identifying common problems in software systems and has proposed various techniques for specifying and detecting anti-patterns in different domains. However, to the best of our knowledge, anti-pattern detection in the domain of Dockerfiles has not been studied yet.

3.6 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 [59].

Search-based techniques [60] treat software refactoring as an optimization problem aimed at enhancing system design quality using software metrics. Ouni et al. [61] developed a multi-objective formulation to address code smells, while Alizadeh et al. [62] employed NSGA-II for dynamically and interactively recommending refactoring solutions.

ML techniques have also been widely used for recommending refactorings. Nyamawe et al. [63] utilized classifiers like LR, CNN, and SVM to suggest refactorings based on historical data and detected code smells. Sagar et al. [64] 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 [65]. Zhang et al. [66] showed LLMs have promising bug-fixing abilities. Madaan et al. [67] used Codex, to improve code readability, while Shirafuji et al.[68] 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 Infrastructure as Code (IaC), particularly for Docker. Srivatsa et al. [69] emphasized the potential of LLMs in generating IaC configurations, suggesting automation can make these processes more accessible.

To the best of our knowledge, there has been no comprehensive study on applying LLMs specifically for Docker-related tasks. Our research explores the automation of Dockerfile refactoring using LLMs and compares its effectiveness to manual refactoring performed by developers.

CHAPTER FOUR

A TAXONOMY OF REFACTORINGS AND TECHNICAL DEBT

4.0 Introduction

Similar to any other complex systems, container-based projects are prone to various quality and technical debt issues related to different artifacts: Docker and Docker-compose files and regular source code ones. Unfortunately, there is a gap of knowledge in how container-based projects actually evolve and are maintained. Few recent studies focused mainly on the detection of the quality issues related to the Dockerfiles in terms of the violation of the basic shell scripts practices [31, 70]. However, there is no holistic understanding of the quality issues of Docker projects that could affect different artifacts beyond just the shell scripts in the Dockerfiles. Furthermore, the correction of these issues via refactorings, defined as changes to improve the structure while preserving the behavior, is still not yet explore in the literature unlike other refactoring domains [71]. As Docker projects become more complex and expensive to maintain [2], it is critical to understand the refactorings that developers would apply.

In this chapter, we address the above mentioned gap by conducting an empirical study on refactorings applied in open-source Docker projects. We analyzed 68 projects, consisting of 19,5 MLOC, along with 193 manually examined commits that include refactorings. The analyzed refactorings were labeled as being performed in Docker and Docker-compose files or regular code files, and related to the Docker-specific debt they alleviated. For the regular code refactorings, we used RefMiner [42] to detect the refactorings in Java files. With the identified refactorings in Docker projects, we developed a refactoring taxonomy. The new knowledge out of the empirical study includes the discovery of (a) the types of technical debt addressed and whether they are specific to Docker projects, (b) the refactoring types

that are common in the different artifacts of Docker projects, and (c) new generalizable Docker-specific refactorings and technical debt categories, if any.

4.1 Approach to Identifying Docker-Specific Refactorings Through Repository Mining

GitHub offers a set of collaborative and social features, as well as versioning and bug tracking mechanisms, which are extensively used in software repository research. All versions of the source code are stored on GitHub, and any modifications are represented by a commit, which includes a written explanation of the change (i.e., commit message). These features enable researchers to mine a large amount of information and various properties related to open-source project practices.

we propose to perform an empirical study based mainly on manual GitHub repository mining to identify the common refactorings in Docker projects. This may present a foundation for new refactoring types for the different artifacts of Docker projects to support practitioners in addressing Docker related technical debts.

The general structure of our approach is sketched in Figure 6.1. We begin by gathering refactoring commits and Docker projects from GitHub, cleaning the data, and selecting a group of projects to evaluate based on a set of criteria. We then move to the Refactoring Identification step, which was carried out separately by four team members. This phase identified both Docker-compose and Dockerfile-related refactoring. In addition, we utilize existing tools to identify source code refactorings that occur within the same commit in which the Docker-compose or Dockerfile was changed. Finally, we divide the newly identified Docker-specific refactoring into various groups depending on the type of technical debt it alleviates. The stages of refactoring identification and classification were reviewed and validated via an inter-agreement coefficient. Each of these steps will be covered in-depth in the next sections.

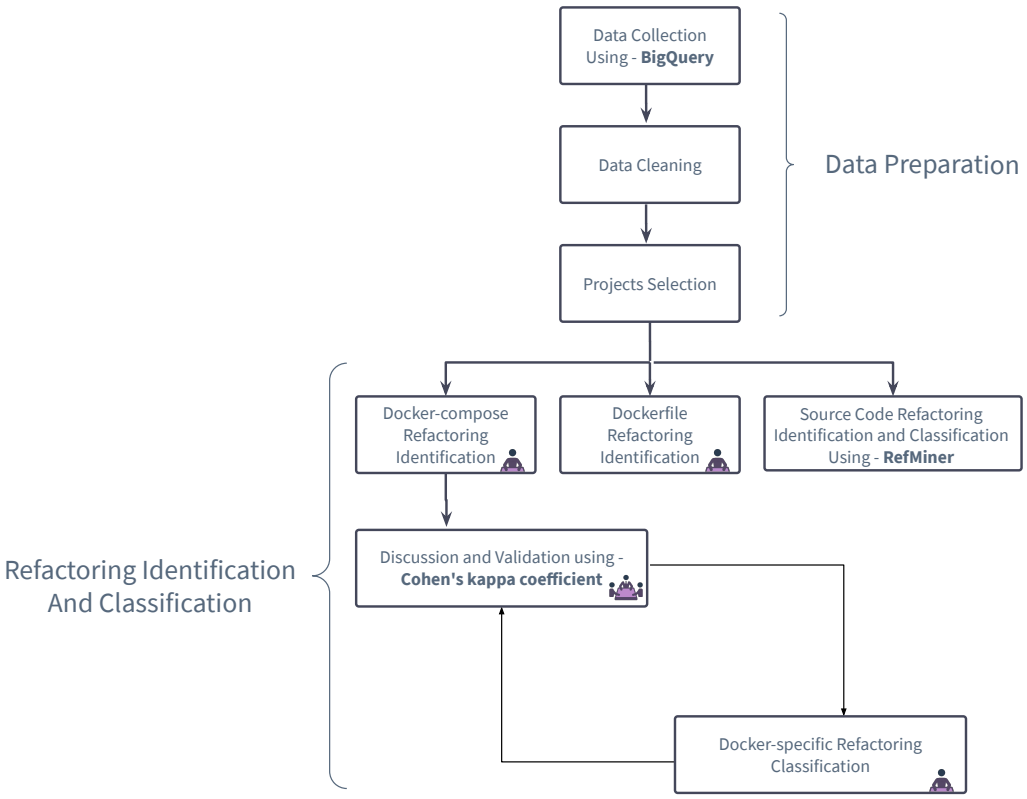


Figure 4.1: Emperical Study Approach for refactoring identification and classification

4.1.1 Data Preparation

We gather Docker projects from the public GitHub archive on BigQuery [72]. BigQuery offers a consistent snapshot of the content of over 3 million open-source GitHub repositories (excluding unnotable projects). We eliminate non-existing projects since BigQuery's last update was in 2019 and removed repositories forked from other repositories to avoid biasing our study, as large and popular projects are forked frequently.

We apply a selection criterion aiming to have at least one commit message mentioning the keywords REFACTOR and DOCKER, and at least one part of the project must include

Docker. To extracted commits that include refactorings from BigQuery achieve, we used the SQL query presented in Listing 3.

```
SELECT * FROM
"bigquery-public-data.github_repos.commits"
WHERE (message LIKE '%refactor%')
AND (message LIKE '%docker%')
```

Listing 3: SQL Query for Extracting Refactoring Commits Related to Docker from BigQuery

The keywords are queried via the SQL like operator, where the % sign is used to represent zero, one, or multiple characters. Since SQL server is by default case-insensitive, the expression %REFACTOR% matches strings containing the word refactor (.e.g.,Refactoring, refactoring, refactored) and the expression %DOCKER% matches strings containing the word docker (e.g., dockerfile, docker-compose).

For each commit, we collect the following properties: repository name, commit Hash/ID, commit date, committer info, commit subject, commit message and code changes performed within this commit. We also use GitHub API to filter commits that didn't include any Docker-related code changes.

The number of commits having the required keywords was initially 1,469 commits, with a maximum of 73 commits per repository. In our final selection, we focus mainly on a total of 68 Docker projects that were mostly written in Java for the code beyond the Docker and Docker-compose files, as it is a popular programming language to develop apps hosting containers [73]. However, we still also consider Docker projects with significant evolution written in other languages, including C++ and Python. To support the manual analysis of the Java code, we used an assisting tool, RefMiner [42]. Still, we note that most of the manual investigation efforts are mainly on the Docker and Docker-compose files to identify relevant Docker-specific refactorings beyond the traditional object-oriented refactoring.

The above selection mechanism yielded a total of 193 commits having the required keywords, ranging from 1 to 12 commits per project and a total of 611 Docker-specific file changes. We manually examine the changes associated with these commits to find patches representing possible refactoring and addressed technical debts.

We examine the changes associated with these commits to find patches representing possible refactoring and addressed technical debts. The identification and classification of these patches are detailed in the next subsection.

4.1.2 Refactoring Identification and Classification

4.1.2.1 Manual Identification Process Since our work represents the first study about Docker-specific refactorings, it is necessary to manually inspect commits for refactoring identification. The keywords matching commits are chosen for manual examination to find patches in Dockerfile and Docker-compose files representing one or more possible refactoring, which requires not only non-trivial efforts but also deep knowledge of the domain. Two of the lab team members are professors and have extensive expertise in refactoring, technical debt, and empirical software engineering. Another team member is an expert in Docker and continuous integration including build repairs. We analyze separately all the considered commits to identify the refactorings and later their rationale. We also discuss the identified refactoring at the end of the process (and not before to avoid any bias) to solidify the results especially when there is disagreements among us.

4.1.2.2 Refactoring Classfication After the refactoring identification phase and in order to understand the refactoring types performed in Docker projects, we analyze the code changes within the selected list to determine the refactoring types and their rationale (e.g., technical debts), whether the refactoring was applied in a Dockerfile or a Docker-compose file or regular source code file (e.g., Java). Discovered refactorings are then organized into a hierarchy based on the addressed technical debts. Hence, a few categories are grouped

beneath distinctive parent categories within the hierarchy, i.e., change-sets containing a few interconnected refactorings are assembled into more common parent categories.

4.1.2.3 Refactoring co-occurrence Since the identified refactoring types of Dockerfile and Docker-compose may impact the regular source of the application hosting the containers, we also manually investigate the selected commits in this study to look at the introduced code changes of Java/C++ files, within the same commit, whenever a Docker-specific refactoring is detected. To support the manual analysis of the Java code, we use an assisting tool, RefMiner [42], a tool used to extract the source code refactoring operations performed between Git commit. We chose RefMiner because it outperformed other technologies in refactoring identification, with a precision of 98 percent and a recall of 87 percent compared to Ref-Finder [74].

4.1.3 Ensuring Refactoring Accuracy

Since we may not be very knowledgeable about the code of the projects as we are not their original developers, we mark the refactorings and their associated commits only when they are very confident that the changes are actual refactorings. We also use commit messages and comments in the code whenever available to confirm their decisions, which is a common practice [75].

Several statistical approaches may be utilized to evaluate the degree of inter-agreement; for this work, we opt for Cohen's Kappa coefficients [76] due to its wide use [43] and support in different statistical tools.

Cohen's Kappa starts by defining a k by k confusion matrix, in which an element f_{ij} defines the number of cases that the first observer assigned a particular case to category i and the second to j. So, f_{jj} is the number of agreements for category j.

$$P_o = \frac{1}{N}\sum_{j=1}^{k} f_{jj}, \tag{4.1}$$

$$r_i = \sum_{j=1}^{k} f_{ij}, \forall i, \text{ and } c_j = \sum_{i=1}^{k} f_{ij}, \forall j, \tag{4.2}$$

$$P_e = \frac{1}{N^2}\sum_{i=1}^{k} r_i c_i, \tag{4.3}$$

N the number of cases / observations

n the numbers of raters / observers

k the number of categories in which a case can be rated

P_o in Equation 4.1 represents the observed proportional agreement, r_i and c_j in Equation 4.2 represent the row and column totals for category i and j, and P_e in Equation 4.3 is a theoretical probability of random agreement computed using observed data to determine the probability of each observer rating each category randomly. The final measure of inter-agreement is given by Equation 4.4.

$$Kappa\_coef = \frac{P_o - P_e}{1 - P_e}. \tag{4.4}$$

$Kappa\_coef$=1 if the raters are entirely in consensus. $Kappa\_coef$=0 if the raters didn't agree on anything different than what has been guessed by chance (P_e). The kappa coefficient might be negative, indicating that there is no real agreement between the two reviewers but rather that the agreement is below random agreement.

We calculate the pairwise Cohen's kappa coefficient between each pair of raters to estimate the inter-agreement. The values of the coefficient in refactoring identification, were between 0.83 and 0.92, which indicates high confidence of agreement. An inter-rater agreement analysis was also used to develop a classification scheme to categorize the

identified refactorings under different technical debt categories. We computed Cohen's kappa coefficient and we reached an average of 0.86 between participants. The few cases of disagreement were discussed at the end of the process and we were able to find a consensus for all of them.

4.2 Current Findings on Docker-Specific Refactorings and Technical Debt Analysis

This section reveals our current findings through two lenses: quantitative analysis and qualitative analysis.

4.2.1 Quantitative Analysis

We manually examine the 193 unique commits. The task of identifying and analyzing refactorings in these commits is highly labor-intensive due to the absence of automated tools. We discover that 99 commits involve Dockerfile-related refactorings, 92 commits include Docker-compose-related refactorings, and 55 commits pertain to regular code refactorings (e.g., Java, C++) induced by changes in Dockerfile or Docker-compose. A total of 38 commits (20%) are identified containing false positives. These false positives arise for several reasons: (i) the term `refactor` is used in commit messages to indicate the need for future refactorings rather than actual refactorings; (ii) refactorings occur outside of Docker-specific contexts; or (iii) the novelty of Docker-specific refactorings in the literature leads to misclassification of regular changes that alter behavior as refactorings.

The Docker-specific technical debts we identify are cataloged in Table 4.1. For Dockerfiles, we identify eight technical debt categories: *Image Size*, *Build Time*, *Duplication*, *Maintainability*, *Understandability*, *Modularity*, *Consistency*, and *Security*. For Docker-compose files, five of these categories are also applicable: *Duplication*, *Maintainability*, *Understandability*, *Build Time*, and *Security*. Additionally, we find *Extensibility* as an additional technical debt category addressed by developers in Docker-compose files.

Table 4.1: Categorization of Docker-Specific Technical Debts

Technical Debt	Artifact	Goal	Situation	Consequence
Image size	Dockerfile	Refactorings are used to reduce the final image size	Adding new/changing components requires huge modifications. It may lead to unexpected behaviors	Huge disk space, difficult to upload, and a huge attack surface.
Build Time	Dockerfile	Refactorings are used to reduce the build or re-build time	Docker engine takes a lot of time to build the final image	Evolving and changing the image became difficult to process.
Extensibility	Docker-compose	Refactorings are performed to increase the level of abstractions and improve the reusability of the Dockerfile and Docker-compose files.	Adding new components require duplication	Duplication.
Duplication	Docker-compose	Refactorings are introduced to fix duplicated fragments in Dockerfile and Docker-compose.	Duplicated fragments	Adding new/changing existing components is difficult and error-prone
Maintainability	Both	Refactorings are performed to ease the modification as well as preventing large impact/spread of future bugs/unexpected behavior	Adding new/changing components requires modifying existing files	Modifications and upgrades require huge endeavors and conceivable downtime
Understandability	Both	Refactorings are applied to reduce the effort to understand code, e.g., renaming elements	Huge efforts to understand code	Slight and simple modifications will become time-consuming
Modularity	Dockerfile	Refactorings are used to reduce image complexity by breaking image into various stages.	Complex image	Adding new/changing existing components is difficult and error-prone
Security	Both	Refactorings are applied to address security-related factors, such as vulnerabilities, weak configurations, or improper access control, to minimize the risk of security breaches and attacks	Image may contain sensitive information or be susceptible to attacks	Security breaches and data leaks
Consistency	Dockerfile	Refactorings are performed to ensure consistency in the naming conventions, use of instruction keywords, or syntax in Dockerfile instructions while taking into account Docker standards and the behavior of the Docker engine	Incorrect use of standards, naming conventions, instruction keywords, or syntax across instructions, and insufficient consideration for Docker engine behavior.	Errors, confusion during the build process, decreased maintainability, and unexpected behavior during the build process.

4.2.1.1 Docker-specific Technical Debts In our analysis of Docker projects, technical debt associated with *Maintainability* emerges as a prevalent focus, with refactorings in this category found in 65 unique commits (33%) across both Dockerfile and Docker-compose files. This trend highlights the objective of container technologies to facilitate system updates with minimal effort and no downtime, often addressing issues like the excessive use of environment variables and TAG updates.

Refactoring in Docker-compose files is primarily aimed at improving *Understandability*, accounting for 44% of the refactorings. This initiative addresses the complexity and potential confusion within Docker configuration files, making them easier to modify and less prone to errors or misunderstandings.

While *Build Time* is less frequently addressed, highlighting the critical role of instruction order in Dockerfiles to optimize caching during rebuilds, it underscores the importance of careful Dockerfile construction.

Modularity is targeted in 7 instances specifically within Dockerfiles, where developers utilize multi-stage builds to separate the build from the runtime environment, thereby reducing unnecessary build dependencies in the final image.

Technical debts related to *Consistency* and *Security* are the least addressed, with only 4 instances each. Refactorings aimed at *Consistency* focus on improving path handling to prevent errors and ensure adherence to Docker standards. In contrast, those related to *Security* seek to mitigate vulnerabilities and enhance access control to improve container safety.

4.2.1.2 Docker-specific Refactoring Types We organize the different refactoring types into a hierarchy based on the technical debt they addressed. Figure 4.2 shows the proposed taxonomy of the refactorings and their rationale. We have also highlighted the number of occurrences of these refactoring types in the analyzed commits. The gray square represents the refactoring type along with the number of occurrences, the blue ellipse represents the artifact (Dockerfile or Docker-compose), and the purple hexagon represents the technical debts that the detected refactoring addressed.

We categorized true-positive refactorings by manually classifying them into 38 new unique refactoring types. Table 4.2 and Table 4.4 show the lists of refactorings that were identified among the analyzed commits. We defined a total of 23 new Dockerfile-related refactoring types and 18 new Docker-compose ones as described in Table 4.2 and Table 4.4.

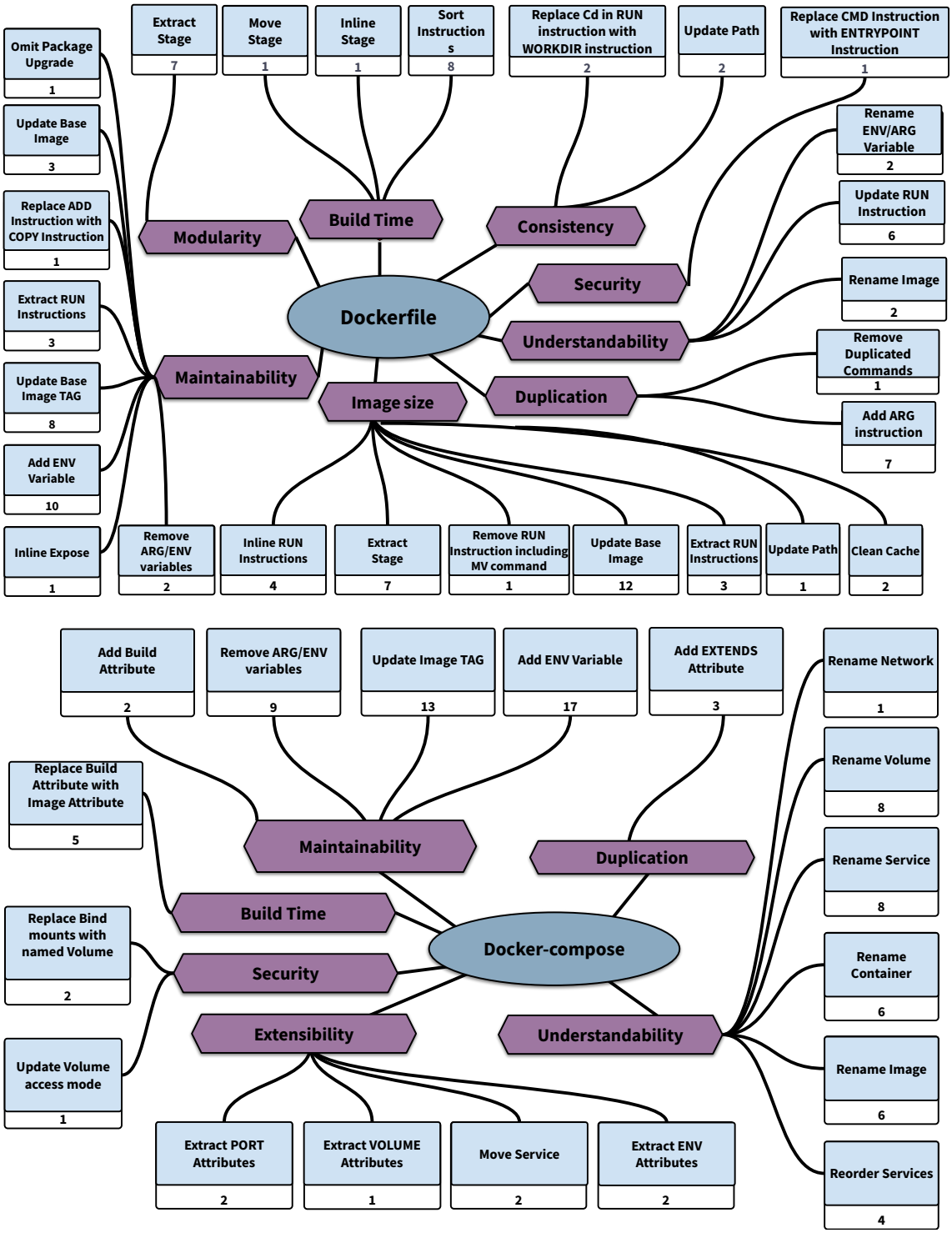


Figure 4.2: Docker specific refactorings taxonomy

The overall number of refactorings identified in the manual commits analysis is 164 to address the aforementioned technical debts.

4.2.1.2.1 Enhancing Maintainability In Docker-related projects, maintainability was a major refactoring theme. This technical debt was addressed by developers in both Dockerfile and Docker-compose files through several identical refactorings. One refactoring involves using environmental variables (27 commits). The other refactoring entails updating the base image TAG in both artifacts (21 commits), and the last refactoring consists of removing environmental and arg variables when they are unused (9 commits).

Besides, we found 5 types of Dockerfile-related refactoring performed to improve maintainability, with a total of 12 commits made across these types.

The first refactoring type, *Update Base Image* (3 commits), consists of using existing images whenever it is possible to avoid building and downloading dependencies on top of the base image. This will reduce maintainability efforts as all required installations are done, and best practices are probably applied, especially when using official images. The second refactoring type *Extract RUN Instructions* (3 commits) consists of extracting a shell script for specific subsequent RUN instructions in the Dockerfile. This refactoring type does not only shrink the image size, but also groups commands in a much cleaner, simpler, and more portable format. The third refactoring operation *Replace ADD Instruction with COPY Instruction* (4 commits) embraces using *COPY Instruction* instead of *ADD Instruction* when files/directories need to be only copied from host to container. It is important to notice that the *ADD Instruction* supports other functionalities, such as the use of URL instead of local files, and it can also recognize the archive format and extract it directly into the destination. These additional functionalities might be considered tricky in practice, as *ADD Instruction* may behave extremely unpredictably. The result of such unreliable behavior often came down to copying when we want to extract and extracting when we want to copy.

Table 4.2: Discovered Docker-specific refactoring types I

Refactoring Type	Artifact	Description
Extract stage	Dockerfile	Extract multi-stage building from a single-stage Building
Inline stage	Dockerfile	Aggregate multi-stage building into a single stage
Move stage	Dockerfile	Move a stage from a multi-stage context into another single stage context in a different Dockerfile
Sort Instructions	Dockerfile	Order Instructions sequence from the least frequently changing to the most frequently changing
Replace ADD Instruction with COPY Instruction	Dockerfile	Replace ADD instruction with COPY Instruction when files/directories need to be only copied from host to container
Extract Run Instructions	Dockerfile	Extract Run instructions commands into a separate script file
Inline Run Instructions	Dockerfile	Inline Run instructions into a single RUN instruction using && operator
Remove Run Instruction including mv command	Dockerfile	Delete RUN instruction with mv command and use previous COPY or ADD instructions to set the correct path
Update Base Image	Dockerfile	Avoid using an unnecessary heavy image, replace a base image with a lighter one
	Dockerfile	Avoid building and downloading dependencies on top of base image; replace the base image with a larger one; if dependencies are fixed, and a similar image exits in a public or private repo, use an existing Image
Rename Image	Docker-compose	Set a relevant Name and TAG including the necessary information of your image (version, software..)
Update RUN Instruction	Dockerfile	Reformat RUN instruction commands; split commands into multiple lines where each line represents a single option/argument, order option/argument alphabetically, use backslash ...
Update Base ImageTAG	Dockerfile	DRY principle, set a dynamic TAG using ARG instruction to avoid creating a new Dockerfile when TAG is only changing
	Dockerfile	Change TAG value or Replace the latest TAG with an explicit TAG
Add ENV variable	Dockerfile	Add ENV variables to store useful system-wide values
Add ARG instruction	Dockerfile	DRY principle, set dynamic instructions using ARG instruction to avoid creating a new Dockerfile for similar images
Extract Ports Attribute	Docker-compose	Extract ports attribute into an override Docker-compose file
Extract Volume Attribute	Docker-compose	Extract volume attribute into an override Docker-compose file
Extract ENV Attribute	Docker-compose	Extract ENV attribute into an override Docker-compose file or use .env file

Table 4.3: Discovered Docker-specific refactoring types II

Refactoring Type	Artifact	Description
Move Service	Docker-compose	Extract service into an override Docker-compose file
Rename Service	Docker-compose	Set a relevant Name for the service
Rename Container	Docker-compose	Set a relevant Name for the container
Rename Volume	Docker-compose	Set a relevant Name for the volume
Add Extends attribute	Docker-compose	Add Extends attribute to inherit configuration from an existing service, thus avoiding duplication
Reorder Services	Docker-compose	Order services based on their dependency order
Update Image TAG	Docker-compose	Change TAG value or Replace the latest TAG with an explicit TAG

Table 4.4: Discovered Docker-specific refactoring types III

Refactoring Type	Artifact	Description
Replace Cd in RUN instruction with WORKDIR instruction	Dockerfile	Define the working directory using WORKDIR instruction and then use relative paths in subsequent RUN instructions, instead of using the 'cd' command.
Update Path	Dockerfile	Modify the paths in RUN, COPY, and ADD instructions to ensure that they are relative to the build context
	Dockerfile	Modify the paths in RUN, COPY, and ADD instructions to ensure that they are relative to the working directory specified in the WORKDIR instruction
	Dockerfile	Modify the paths in COPY instructions to avoid copying unnecessary files
Clean Cache	Dockerfile	Clear any cached data or artifacts that may have been generated by previous build steps using commands like 'yarn clean cache', 'npm cache clean', or 'apt-get clean'.
Omit Package Upgrade	Dockerfile	Remove any 'apt-get-upgrade' or similar commands used for package upgrading.
Rename ENV/ARG Variable	Dockerfile	Set a relevant name for ARG or ENV variables
Inline Expose	Dockerfile	Inline EXPOSE instructions into a single EXPOSE instruction
Replace CMD Instruction with ENTRYPOINT Instruction	Dockerfile	Replace CMD Instruction with ENTRYPOINT Instruction, which sets the command that will be run when the container starts.
Remove Duplicated Commands	Dockerfile	Remove any duplicated commands inside RUN instructions.
Remove ARG/ENV Variables	Dockerfile	Remove unused ARG/ENV variables
	Docker-compose	
Rename Network	Docker-compose	Set a relevant name for a network
Add Build Attribute	Docker-compose	Add the Build attribute to specify the build context and Dockerfile location.
Replace the Build Attribute with the Image Attribute	Docker-compose	Replace the Build attribute with the Image attribute when the image already exists to avoid rebuilding.
Update Volume Access Mode	Docker-compose	Change the volume access mode from read-write to read-only
Replace Bind Mounts with Named Volume	Docker-compose	Replace the bind mounts storage option in a Docker Compose file with the named volume storage option.

The fourth type of refactoring *Omit Package Upgrade* (1 commit) involves removing any package upgrade commands from the Dockerfile to minimize the risk of introducing unpredictability and potentially breaking the application inside the container. In fact, these commands can cause issues by upgrading dependencies that are not compatible with the application or by introducing bugs that could be difficult to track down. By avoiding package upgrades, the Dockerfile can ensure that the container is always built with a stable set of dependencies.

4.2.1.2.2 Reducing Image Size Unsurprisingly, a large proportion of the analyzed refactorings in Dockerfile aimed at reducing the image size (36%). We found seven possible refactorings targeting this technical debt, namely, *Extract RUN Instructions* (3 commits), *Inline RUN Instructions* (4 commits), *Remove RUN Instruction including MV command* (3 commits), *Update Path* (2 commits), *Clean Cache* (2 commits),*Update Base Image* (12 commits) and *Extract stage* (7 commits).

Remove RUN Instruction including MV command, in particular, aims to reduce the number of layers in an attempt to shrink the image size by removing the RUN instruction with MV command and using previous COPY or ADD instructions to set the correct path, if possible. *Clean Cache* is the process of running cleanup commands (e.g., apt-get clean, yarn clean) to remove unnecessary files and data, resulting in lower image size and more efficient use of storage space. *Extract Stage* embraces applying multi-stage building by separating build from the run-time environment, which not only ensures *Modularity* but also helps avoid the inclusion of unnecessary build dependencies in the final image, besides maintainability. *Update Base Image* can also be applied to reduce the image size. In such scenarios, developers avoid downloading unnecessary dependencies by replacing the base image with a lighter version, including only the required dependencies.

Regarding *Update Path*, this refactoring aims to reduce the size of the final Docker image by only copying the necessary files to the container during the build process. This is

achieved by carefully selecting the paths used in the COPY instruction to exclude any files or directories that are not required for the application to function properly.

4.2.1.2.3 Promoting Consistency In addition to reducing the image size, *Update Path* refactoring also focuses on improving *consistency*. By ensuring that the paths used in RUN, COPY, and ADD instructions are relative to either the build context or the working directory specified in the Dockerfile's WORKDIR instruction, the refactoring helps avoid issues that can arise from using absolute paths or inconsistent paths. Having inconsistent paths in Dockerfiles can lead to errors during the build process. For example, if a COPY instruction specifies a file path that does not exist in the build context, the build will fail with an error. Similarly, if a RUN instruction attempts to execute a script or command from a directory that does not exist.

Consistency technical debt was also addressed by *Replace Cd in RUN instruction with WORKDIR instruction* (2 commits). In fact, when using the "RUN cd" instruction in a Dockerfile, you are essentially changing the current working directory (CWD) for subsequent instructions within that layer. However, this can lead to inconsistencies in the Dockerfile, as different layers may have different CWDs. Using the "WORKDIR" instruction, on the other hand, allows you to set a consistent working directory across all layers of the Dockerfile.

4.2.1.2.4 Improving Understandability *Understandability* is the major refactoring category in Docker-compose files. In this category, 75% of the refactorings were performed to improve naming. The observed renaming had a variety of motivations, such as avoiding the usage of irrelevant names (e.g., using the app as a service name), including software information (e.g., software version) in the image and TAG names, as well as keeping naming consistency throughout the project. The renaming was also present in Dockerfile (4 commits). We also found several refactorings applied to make the Dockerfile less confusing and more understandable. Such changes involved the improvement of structural aspects

as a way to address the *understandability* technical debt. For example, *Update RUN Instruction* (6 commits) in Dockerfile, which consists of formatting commands within RUN instructions (e.g., splitting commands into multiple lines where each line represents a single option/argument) and *Reorder Services* (4 commits) in Docker-compose where developers reordered the sequence of the services based on their dependency order.

4.2.1.2.5 Optimizing Build Time Unlike *Extract Stage*, *Inline Stage* (4 commits) and *Move Stage* (3 commits) aim at aggregating multi-stage building into a single stage when multi-staging is unnecessary (i.e., no significant dependencies) and extracting a stage from a multi-stage context into a new single-stage context (e.g., extracting test stage into a new Dockerfile), respectively. The reason behind these refactorings is to improve the *build time* when multi-staging can be avoided. As multi-staging requires building intermediary images that significantly affect the *build time*.

Replace Build attribute with Image attribute (5 commits), refactoring was also introduced to address *Build Time* technical debt. In point of fact, When the build attribute is specified in docker-compose, it instructs Docker to build the image from the specified Dockerfile. This can add to the build time, especially if the Dockerfile is large or contains many dependencies that need to be installed. On the other hand, if the build attribute is not specified and the image attribute is added, the docker image will be pulled from a remote registry or locally, and the build time will be thus reduced as Docker does not need to rebuild the image.

4.2.1.2.6 Eliminating Duplication We also found in Docker-project eight Dockerfile commits aiming at improving code design by avoiding *duplication* and fostering the reuse of the code fragments. *Add ARG instruction* (7 commits) in Dockerfile involves adding arguments using the ARG instruction to define common identifiers (e.g., software version and file/directories paths) that can be later used inside other instructions such as COPY, ADD, and RUN, leading to a dynamically changing Dockerfile, as these identifiers can be affected at build time. *Remove Duplicated Commands* (1 commit) in Dockerfile attempts

to eliminate any repeated commands from RUN instructions, even if the duplication is not explicit, such as when a command includes a nested call to another command, which was then called again in a RUN instruction.

Besides, *duplication* in Docker-compose was reduced using configuration inheritance using extended attribute refactoring (3 commits).

4.2.1.2.7 Enhancing Extensibility In addition, eight Docker-compose-related commits have implemented refactoring to address the technical debt associated with *Extensibility* in order to make the configuration code more generalizable, reusable, and inter-operable by moving attributes such as PORT, VOLUME, and ENV, as well as moving services to an override Docker-compose file. Such practice helps developers reuse a single Docker-compose file across development and production while being able to run different services.

4.2.1.2.8 Bolstering Security Practices Furthermore, We identified 3 types of Docker-compose refactoring that address the *Security* technical debt, namely: *Replace CMD Instruction with ENTRYPOINT Instruction* (1 commit),*Update Volume access mode* (1 commit), and *Replace Bind mounts with named Volumes* (2 commits).

The *Replace CMD Instruction with ENTRYPOINT Instruction* refactoring is designed to enhance the security of Docker images by specifying the command that should be executed when the container starts. As a point of fact, the CMD instruction in Dockerfiles is used to specify the default command and parameters that will be executed when the container starts. This command and its parameters can be overridden from the Docker CLI when running a container. On the other hand, the ENTRYPOINT instruction sets the command and parameters that will be executed when the container starts, and it does not allow these to be overridden by CLI parameters. This means that the command and parameters specified in the ENTRYPOINT instruction will always be executed when the container starts, providing

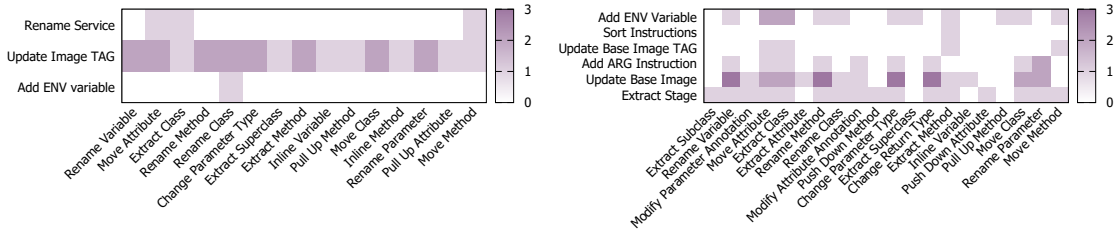
a more secure way to run containers. This helps to prevent attackers from injecting malicious commands within the container.

The *Update Volume access mode* refactoring is focused on setting the appropriate access mode for volumes in Docker-compose files. By ensuring that volumes are set to read-only when appropriate, this refactoring can prevent attackers from modifying the contents of the volume.

The *Replace Bind mounts with named Volumes* refactoring involves using named volumes instead of bind mounts in Docker-compose files. This change provides a more secure method of accessing data files by isolating the volume from the host system. Thus, the contents of the volume are protected from modification by attackers who might have gained access to the host system. Additionally, named volumes are used to provide a more robust solution for accessing data files than bind mounts, as they are managed by Docker and can be securely shared across containers.

4.2.1.2.9 Refactorings Co-evolution Among the analyzed commits, we found 31 regular code refactoring cases performed along with Docker-related refactoring operation, Figure 4.3 shows two heat-maps illustrating the co-occurrence of both refactoring types. Docker-related refactoring types are represented on the vertical axis and regular code refactorings are represented on the horizontal axis. Each cell in the heatmap indicates the occurrence of a Docker-related refactoring type along with a regular code refactoring type. Darker colored cells indicate a strong co-occurrence frequency while a lighter color signifies a weaker co-occurrence. The observed refactorings included 22 regular code refactoring types along with 9 Docker-related refactoring types, 3 for Docker-compose (a) and 6 for Dockerfile (b). It is clear that *Update Image TAG*, *Update Base Image*, and *Extract Stage* are associated with extensive code refactoring of the hosting application involving almost all the refactoring types. In fact, those Docker-specific refactorings directly impact the code hosting

the containers as they introduce significant changes in the Image or the Stage(s), which are typically called from the hosting code similar to functions in APIs in other contexts.



(a) Docker-compose. (b) Dockerfile.

Figure 4.3: Refactorings co-evolution: regular code and Docker-specific refactorings

4.2.2 Qualitative Analysis

The qualitative analysis aims at obtaining and analyzing some examples of commits where the Docker-specific refactorings were found to address the most common/important technical debt issues discussed in the previous section.

4.2.2.1 Dockerfile Technical Debt and Refactoring Examples Several of the analyzed refactorings aimed at improving *maintainability* from several perspectives. Let us consider the example of the *Update Base Image* refactoring shown in Listing 4. In this listing, `openjdk` was initially used as a base image (Line 5), and `Dockerize` software was installed using RUN instructions (Lines 7–12). However, the official `Dockerize` image already exists with the required version v0.6.1, and it can be easily pulled from DockerHub to act as a starting point for the Dockerfile (Line 6). This can save a lot of time spent on maintenance because all the installation steps are done, and official images can be highly trusted.

If we consider another example of refactoring that addresses maintainability, as shown in Listing 5. In this commit, the developers applied the *Inline Expose* refactoring technique by combining multiple `Expose` instructions (lines 7-22) into a single instruction (line 29).

```
diff --git Dockerfile
--- a/docker/elasticsearch/Dockerfile
+++ b/docker/elasticsearch/Dockerfile
@@ -1,17 +1,12 @@
-FROM openjdk:8
+FROM jwilder/dockerize:0.6.1
-RUN apt-get update && apt-get install -y wget
-     && apt-get install -y curl
-ENV DOCKERIZE_VERSION v0.6.1
-RUN wget https://github.com/jwilder/docke[...]
-    && tar -C /usr/local/bin -xzvf docker[...]
-    && rm dockerize-linux-amd64-$DOCKERIZ[...]
```

Listing 4: Commit `4f221f9` from `datahub`: Refactored Dockerfile to reduce the number of pushed layers on a rebuild.

The reason for this refactoring was to avoid potential errors that could arise during image creation or container execution on different machines due to having too many layers of size 0. The commit message explicitly states the goal of the refactoring: "*Refactored docker images to avoid 127 layers/parents possible error on some machines.*"

4.2.2.2 Docker-compose Technical Debt and Refactoring Examples Regarding the Docker-compose files, *Extensibility* is a common and frequent technical debt as described in the quantitative analysis. In general, the Docker-compose file's extensibility should be improved when both development and production environments are located in the same file. *Move Services* refactoring shown in Listing 8 can be used to improve the extensibility of Docker compose files. In this example, developers moved `benchmark_resin` services (Lines 9–12) to a new override file (Line 16). Therefore, they were able to reuse a single Docker-compose file while being able to run different services.

We have also found that the inheritance was mainly improved in Docker-compose files to remove *Duplication*. The *Add Extends Attribute* refactoring presented in Listing 7 is a common refactoring type for Docker-compose to address duplication issues. This refactoring helped to solve the configuration duplication issue by using the `extends` attribute (Line 12) and specifying the parent service `db`(from the parent Docker-compose file).

```
diff --git Dockerfile
--- a/nodes/DockerfileMaster
+++ b/nodes/DockerfileMaster
@@ -14,18 +14,12 @@

-# SSH and SERF ports
-EXPOSE 22 7373 7946
-
-# HDFS ports
-EXPOSE 9000 50010 50020 50070 50075 50090 50475
-
-# YARN ports
-EXPOSE 8030 8031 8032 8033 8040 8042 8060 8088 50060
-
-# Zookeeper ports
-EXPOSE 2181 2888 3888
-
-# Accumulo ports
-EXPOSE 9999 9997 50091 50095 4560 12234
-
-# Spark ports
-EXPOSE 8080 8081 7077 4040 4041 18080
+# SSH and SERF ports 22 7373 7946
+# HDFS ports 9000 50010 50020 50070 50075 50090 50475
+# YARN ports 8030 8031 8032 8033 8040 8042 8060 8088
+# Zookeeper ports 2181 2888 3888
+# Accumulo ports 9999 9997 50091 50095 4560 12234
+# Spark ports 8080 8081 7077 4040 4041 18080
+EXPOSE 22 7373 7946 9000 50010 50020 [ ...] 4041 18080
```

Listing 5: Commit `fd6b0db` from `lagoon`: Refactored Dockerfile to reduce image layers and avoid possible errors in some machines.

```
diff --git docker-compose.yaml
--- a/docker-compose.yaml
+++ b/docker-compose.yaml
@@ -24,22 +24,3 @@ services:
-benchmark_resin:
-  build: {[...]}
-  depends_on: [...]
-  ports: [...]
diff --git/hexagon_benchmark/docker-compose.yaml
--- /dev/null
+++ b/hexagon_benchmark/docker-compose.yaml
```

Listing 6: Commit `99109b6` from `hexagon`: Refactored Docker-compose file to make "benchmark services" optional.

```
diff --git a/docker-compose-jasper.yml
--- a/docker-compose-jasper.yml
+++ b/docker-compose-jasper.yml
@@ -1,105 +1,61 @@
db:
-  image: muccg/postgres-ssl:9.4
-  environment:
-    - POSTGRES_USER=rdrfapp
-    - POSTGRES_PASSWORD=rdrfapp
-  ports:
-    - "5432"
+  extends:
+    file: docker-compose-common.yml
+    service: db
```

Listing 7: Commit `f7d2b4d` from `rdrf`: Refactored Docker-compose file to remove duplicated configuration.

4.3 Threats To Validity

In this study, we selected 68 Docker projects to identify refactoring types and technical debt categories but they may not be representative of the very large number of Docker projects on GitHub. To address this threat, we ensured that the selected projects are diverse in domains, sizes, and realistic in terms of evolution history, and popularity. Thus, we used various GitHub metrics including the number of contributors, users, stars, commits, and other relevant factors, to evaluate their popularity and diversity. Despite Java being the dominant language on applications hosting containers (chosen to facilitate the manual analysis of the regular code refactoring), around 30% of the selected projects are written in C++.Indeed, the refactoring types (and their rationale) in object-oriented programming are almost the same, thus the impact of our selection criteria are limited to the generalizability of the co-evolution results between Docker and Docker-compose refactorings, and regular code ones.

The manual identifications and classification of refactoring types and technical debt categories can be subjective. To mitigate this threat, four experts evaluate all the selected commits separately and the agreement coefficient score between all of them was high for all

the identification and classification results. For the few cases of disagreement, the different experts discussed the commit(s) after submitting their results to avoid any bias. Our study also involved many hours of manual inspection and analysis to understand and categorize the Docker-specific refactorings. To mitigate these threats, the four experts used the commit messages, pull-requests descriptions, and comments in the code to better understand the context of the changes and they only marked the refactorings and their rationale when they are very confident about them. For the regular code refactorings (co-evolution study), RefMiner [42] was used to confirm and aid the manual inspection of the refactoring types, even though all commits were manually inspected carefully for any potential false positives.

4.3 Conclusion

This chapter introduces a study that seeks to advance the knowledge of Docker refactorings and technical debts, which are under-explored in the literature. We have manually investigated and defined specific refactoring types and technical debt categories for Docker projects. We have also studied the co-evolution between applying those Docker-specific refactorings and regular refactorings on the code of the app hosting the containers. A taxonomy of refactorings in Docker projects was proposed, including 23 new Dockerfile-related refactorings, 18 Docker-compose-related refactorings, and 9 technical debt categories.

Our study's analyses of Docker refactorings and technical debts have led to several key results. For Dockerfiles, we identified that improvements in build time, maintainability, and image size reduction are primary goals, evident in refactorings like *Update Base Image* and *Sort Instructions*. In Docker-compose files, the focus shifts to enhancing reusability, understandability, security, and extensibility, as seen in refactorings like *Move Services* and *Add Extends Attribute*. Significantly, we observed a co-evolution of Docker and Docker-compose file refactorings with source code refactorings in Docker projects. This

indicates a synergistic relationship between Docker configurations and application code evolution, necessitating an integrated approach to system maintenance and evolution.

CHAPTER FIVE

DRMINER TOOL FOR IDENTIFYING AND ANALYZING REFACTORINGS

5.0 Introduction

Dockerfile developers frequently revisit their previous contributions in pursuit of clarity and improvements [14]. The extraction and comprehension of these manual refactorings, particularly from Dockerfiles or related configurations, entail various implications, including **(1) Error Detection and Prevention:** Every change in a Dockerfile, regardless of size, has the potential to introduce bugs or issues [77]. Thus, inaccuracies during the refactoring process can result in unexpected behaviors in Docker containers, which could compromise the functionality, security, and maintainability of applications. Early identification and management of these risks are essential for preventing problems from escalating. **(2) Understanding and assessing Refactoring:** Refactorings are motivated by specific objectives. Comprehending the underlying rationale behind these changes is crucial. This understanding aids developers in ensuring that the refactoring aligns with the intended purpose. This understanding is equally valuable for code reviewers, providing critical context and leading to more informed decisions [78]. **(3) Interdependencies in Refactoring Activities:** Dockerfile refactorings do not occur in isolation but are interwoven with broader changes in the project, including updates to related code or configurations. Developers need to recognize these interdependencies to proactively address and manage potential cascading changes. In fact, revisions that involve co-changes with Dockerfiles constitute over 70% of all revisions [14]. **(4) Refactoring Impact on Quality Metrics:** The effects of Dockerfile refactorings extend beyond immediate code improvements, impacting broader software quality metrics such as build time, image size, container performance, and resource efficiency.

Evaluating the impact of refactorings on these metrics is key to deriving actionable insights that can optimize current configurations and guide future refactoring strategies [3].

Considering all these implications, as well as the fact that developers often overlook the documentation of their refactoring activities and frequently merge them with other modifications [3, 14], automatically detecting that a configuration segment has been refactored is invaluable in the context of Docker. While several refactoring detection tools exist, catering predominantly to Java, with some extending to JavaScript, C, and Python [16, 17, 18, 19], they are not suitable for Dockerfiles. This is due to Dockerfiles' unique Infrastructure as Code (IaC) nature and declarative syntax, which significantly differ from traditional source code. Additionally, the types of refactorings in Dockerfiles are distinct, further necessitating specialized tools for their detection and analysis.

To bridge this void, In this chapter, we introduce DRMiner, a tool tailored for the identification and analysis of Dockerfile-specific refactoring. DRMiner analyzes Dockerfile refactorings by first building models of two Dockerfile revisions linked to a commit. It then matches components within these models, identifying 12 types of refactorings based on a predefined set of rules derived from the refactoring definitions of chapter IV.

5.1 Approach to Constructing DRMiner

As illustrated in Figure 7.1, *DRMiner* takes as input two revisions of a Dockerfile, denoted as v_1 and v_2, corresponding to a specific commit and its preceding version. The tool's first step is to construct two models, T_1 and T_2, which represent the Dockerfile content at these revisions (see subsection 5.1.1). Following model construction, *DRMiner* enters its component matching phase detailed in subsection 5.1.2. During this phase, the tool systematically aligns stages and instructions between the Dockerfile models T_1 and T_2. *DRMiner* then applies a set of detection rules using matched and unmatched components. We develop these rules based on the refactoring definitions established in previous chapter.

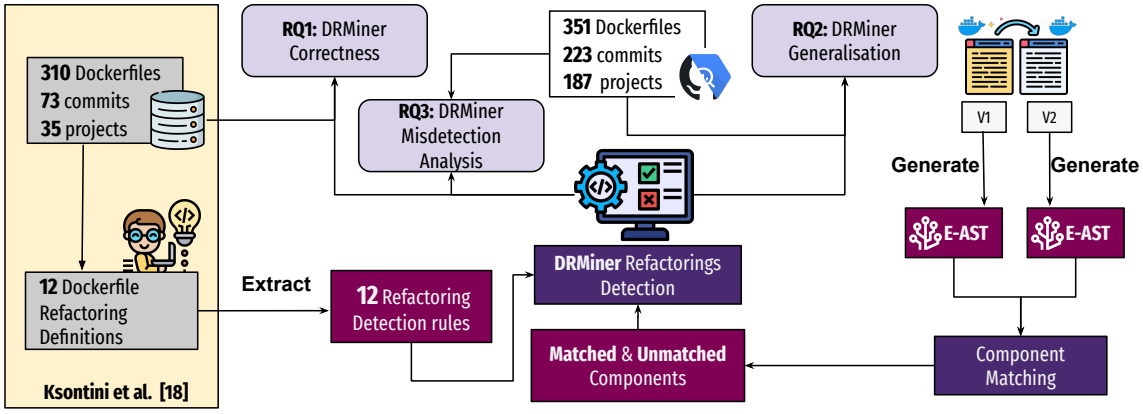


Figure 5.1: DRMiner approach overview

5.1.1 Dockerfile Content Representation and E-AST Generation

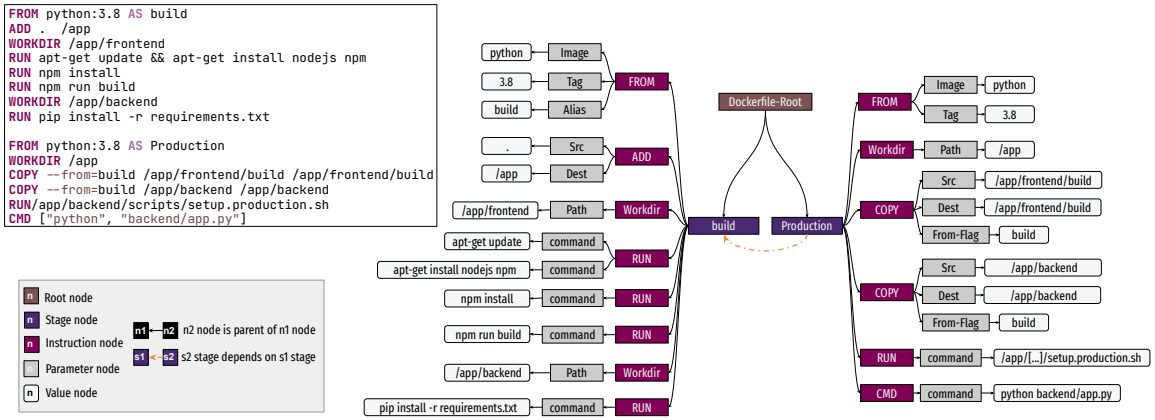


Figure 5.2: Representation of a Dockerfile example using E-AST

The tool first intend to construct a comprehensive representation of the Dockerfile's content across two given revisions. For this, we go through a Dockerfile content analysis phase and aim to capture its structural and semantic nuances, thereby facilitating the subsequent identification of refactorings. This could be achieved without the intricacies of the actual syntax using Abstract Syntax Trees (ASTs) [79], a concept that has been previously used in the context of Dockerfiles by Henket et al. [31]. Despite laying significant

groundwork, the existing version of AST has notable gaps. For instance, while it effectively captures the subtleties of embedded scripting within Dockerfiles, it tends to overlook the equally crucial structural complexities of instruction and stages. We present *"E-AST"*, an Enhanced AST version that fills this gap and guarantees the following needs:

(1) Comprehensive Flag Representation: In a Dockerfile, certain instructions are accompanied by flags that can significantly modify their functionality. While the *AST* excludes all flags, the *E-AST* ensures a comprehensive flag representation, as exclusions in the AST may distort the intended purpose of the Dockerfile.

(2) Multi-stage Build Differentiation: While the *AST* doesn't differentiate between multi-stage builds, a fundamental aspect of Docker's modular design, the *E-AST* distinctly represents each stage and its interdependence. This ensures that interdependencies and data exchanges among different stages are clearly captured.

(3) Order Preservation: The *E-AST* maintains the order of instructions as they appear in the Dockerfile, a crucial factor influencing the build process and the resultant container image, addressing the limitation of standard ASTs in capturing this essential contextual information.

The process of constructing the E-AST involves the following steps:

Step 1: **Establishing the Dockerfile Grammar:** The foundation of any structured analysis begins with a clear understanding of the language's grammar. For Dockerfiles, this involves a meticulous review of the official Docker documentation [80], valid up to version 1.6.0. We note each instruction, its syntax, parameters, flags, and associated behaviors. Based on this review, we define a formal grammar for Dockerfiles using the Backus-Naur Form (BNF) [81]. This grammar captures the structure of Dockerfile instructions, their ordering constraints, and the expected format of their arguments and flags. To illustrate, consider the COPY instruction in Dockerfiles. Its grammar can be represented as:

```
1 <COPY> ::= "COPY" <flag>* <src>+ <dest>
```

```
<flag> ::= ("--from=" | "--chown=" | "--chmod=") <str> | "--link"
<dest> ::= <str>
<src> ::= <str>
<str> ::= /([^$\n]|(\$<var>|\${<var>}))*/
<var> ::= /[a-zA-Z_][a-zA-Z0-9_]*/
```

Beyond the structural grammar, Dockerfiles also have semantic rules, such as the requirement for the FROM instruction to always precede other instructions, except if it is an ARG instruction. These rules are extracted as additional constraints that complemented the formal grammar. This phase ensures that the foundation upon which subsequent analyses would operate is robust and accurate.

Step 2: **Preprocessing:** Before analysis, we exclude Dockerfiles with unresolved merge conflicts or missing/empty content. Additionally, we remove comments and rectify any formatting inconsistencies, such as varying indentation and unnecessary whitespace.

Step 3: **Tokenization and Syntax Validation:** In the parsing phase, we use a Dockerfile parser[1] to split each instruction into key-value pairs, where the key is the instruction type and the value is its textual content. Then, our custom-developed parser tokenizes this textual content based on the established grammar, using predefined patterns, often formulated as regular expressions. Any deviations from the expected structure are flagged immediately, and the concerned Dockerfile will be discarded. We note that for the RUN instruction, each command within it is treated as a separate token, while script execution is represented as a script path.

Step 4: **Capturing Stages**: The linear sequence of tokens is fed into a parser that identifies the start of a new stage with every FROM instruction encountered. As it progresses, the parser groups tokens into logical blocks corresponding to each Docker instruction. These blocks, or instruction groups, are then associated with their respective stages.

[1] https://pypi.org/project/dockerfile-parse/

Step 5: ***E-AST* Generation** This phase aims to transform the segmented Dockerfile representation into a structured, hierarchical format. As illustrated in Figure 5.2, the entire Dockerfile is encapsulated by the root node. Branching out from this root, a distinct node is crafted for each stage. Nested within these stages are secondary nodes, each symbolizing a specific Docker instruction present within that stage. The *E-AST* is constructed in a manner that inherently preserves the order of instructions as they appear in the Dockerfile. This is achieved by sequentially traversing the tokenized instructions and constructing corresponding nodes in the *E-AST* in the same order.

Step 6: **Stages Dependency Mapping:** This step involves identifying dependencies between Dockerfile stages. Dependencies occur when one stage refers to another stage, typically achieved through the use of the COPY instruction with the *–from* flag or via the FROM instruction. As illustrated in Figure 5.2, a clear dependency is observed between the *production* and the *build* stages. Specifically, the production stage references the build stage twice using the COPY instruction with the *–from=build* flag. This creates a dependency link in the *E-AST*, connecting the nodes representing the *build* and *Production* stages.

In order to identify refactorings within a commit, DRMiner generates E-ASTs (T_1 and T_2) for each Dockerfile revision present in the commit. Initially, Dockerfiles that are newly created within the commit are screened out, but they can be revisited if needed for specific refactoring scenarios.

5.1.2 Component Matching

The matching process, involves pairing Dockerfile components, either stage-to-stage or instruction-to-instruction. Our component matching algorithm, as outlined in Algorithm 5.1, is inspired by [42]. The similarity lies in performing the matching in a bottom-up fashion, starting from matching instructions and then proceeding to composite instructions forming a stage, all without the need to define similarity thresholds.

Algorithm 5.1: Component Matching Algorithm

Input: Trees T_1 and T_2

Output: Set MI of matched instruction pairs, MS of matched stage pairs. Sets of unmatched instructions and unmatched stages UI_1, UI_2, US_1, US_2

(1) $MI, MS \leftarrow \emptyset$

(2) $T1 \leftarrow \text{preprocess}(T_1)$, $T2 \leftarrow \text{preprocess}(T_2)$

(3) $UI_1 \leftarrow$ T1.instructionNodes, $UI_2 \leftarrow$ T2.instructionNodes

(4) $US_1 \leftarrow$ T1.stageNodes, $US_2 \leftarrow$ T2.stageNodes

(5) $UI_1, UI_2 \leftarrow \text{matchComponentsRound1}(UI_1, UI_2, MI)$

(6) $UI_1, UI_2 \leftarrow \text{matchComponentsRound2}(UI_1, UI_2, MI, \text{"instr"})$

(7) $US_1, US_2 \leftarrow \text{matchComponentsRound1}(US_1, US_2, MS)$

(8) $US_1, US_2 \leftarrow \text{matchComponentsRound2}(US_1, US_2, MS, \text{"stage"})$

(9) **Function** `matchComponentsRound1`(N_1, N_2, M)**:**

(10) **for** *each component* c_1 *in* N_1 **do**

(11) **for** *each component* c_2 *in* N_2 **do**

(12) **if** $c_1.tree = c_2.tree$ **then**

(13) $M \leftarrow M \cup \{(c_1, c_2)\}$

(14) $N_1 \leftarrow N_1 \setminus \{c_1\}$

(15) $N_2 \leftarrow N_2 \setminus \{c_2\}$

(16) **break**

(17) **end**

(18) **end**

(19) **end**

(20) **return** N_1, N_2

(21) **Function** `matchComponentsRound2`$(N_1, N_2, M, type)$**:**

(22) $Scores \leftarrow \emptyset$

(23) **for** *each component* c_1 *in* N_1 **do**

(24) **for** *each component* c_2 *in* N_2 **do**

(25) $score \leftarrow \text{computeScore}(c_1, c_2, \text{type})$

(26) **if** $score > 0$ **then**

(27) $Scores \leftarrow Scores \cup \{(c_1, c_2, score)\}$

(28) **end**

(29) **end**

(30) **end**

(31) **while** *Scores is not empty* **do**

(32) $MaxScorePairs \leftarrow \text{getMaxScorePairs}(Scores)$

(33) $(c_1^*, c_2^*) \leftarrow \text{findBestMatch}(MaxScorePairs, \text{type})$

(34) $M \leftarrow M \cup \{(c_1^*, c_2^*)\}$

(35) $N_1 \leftarrow N_1 \setminus \{c_1^*\}$

(36) $N_2 \leftarrow N_2 \setminus \{c_2^*\}$

(37) Remove all pairs containing c_1^* or c_2^* from $Scores$

(38) **end**

(39) **return** N_1, N_2

(40) **Function** `computeScore`$(c_1, c_2, type)$**:**

(41) **if** $type =$ *"instr"* **then**

(42) **return** $\frac{\text{commonChilds}(c_1, c_2)}{\text{totalChilds}(c_2)}$

(43) **else**

(44) **return** $\frac{\text{matchingInstructions}(c_1, c_2, MI)}{\text{totalInstructions}(c_2)}$

(45) **end**

(46) **Function** `findBestMatch`$(P, type)$**:**

(47) **if** $type =$ *"instr"* ***and*** *pairs in P have common* i_1 *or* i_2 **then**

(48) $P \leftarrow \arg\min_{(i_1, i_2) \in P} \text{Distance}(\text{uncommonChilds}(i_1, i_2))$

(49) **end**

(50) **return** first pair of P

To minimize the likelihood of incorrect matches, we adopt a conservative approach. This involves conducting matching in multiple rounds, with each subsequent round employing a less strict match condition than the preceding one. As a result, components matched in earlier rounds are considered more reliable and are excluded from being matched in subsequent rounds. This approach reduces the number of component combinations to check in each successive round.

Algorithm 5.1 takes as input two *E-ASTs*, T_1 and T_2 representing the nesting structure of the Dockerfile (as shown in Figure 5.2) from the parent and child commits, respectively. At the highest level, each *E-AST* encapsulates the entire Dockerfile as a grand tree. Within this overarching structure, individual stages of the Dockerfile can be visualized as major subtrees, with each stage's unique identifier serving as its root. Further nested within these stage subtrees are instruction subtrees. Each instruction is represented by its type as the root, with its parameters branching out to the leaves.

Instruction-to-Instruction Matching We match instructions in two rounds (line5-6). During the initial matching round, we pair instructions based on an identical tree representation, signifying a complete structural and content alignment between them, from the root down to the finest branches and leaves. However, as the algorithm progresses to the second round, as outlined in the `matchNodesRound2` function, the matching becomes slightly more relaxed. Here, the algorithm intends to capture not just identical but also closely related instructions, thereby broadening the scope of potential matches.

The `computeScore` mechanism is a key component of this second round. Rather than adopting a simplistic binary framework of matching or mismatching, this mechanism employs a scoring system to quantify the similarity of instructions. The score is determined by calculating the ratio of shared child elements between two instructions to the total number of child elements in the second instruction. After calculating the scores for all possible pairs of instructions, the algorithm focuses on the pairs that have the highest score. However,

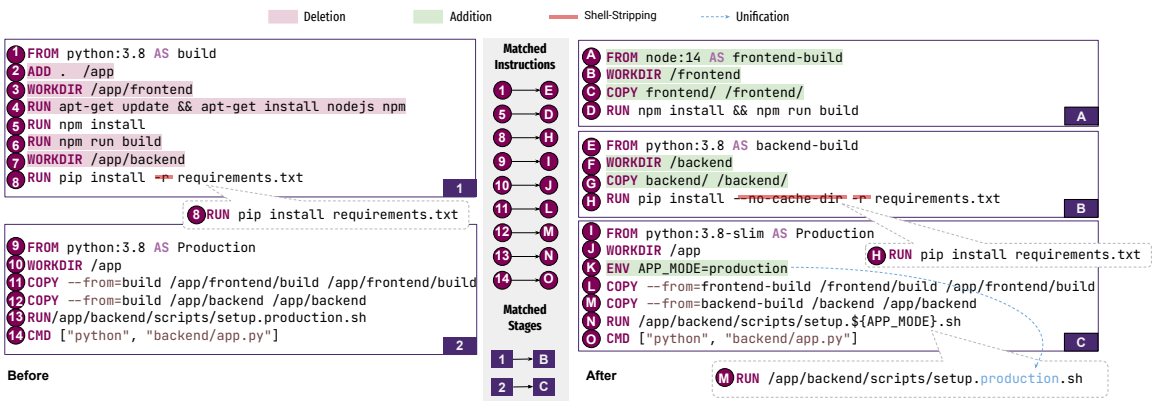


Figure 5.3: Dockerfile component matching example

difficulties arise in situations where multiple pairs have the same highest score and also share a common set of instructions. In order to tackle this issue, the algorithm incorporates a refinement phase (function `findBestMatch`). Instead of randomly choosing a pair with a high score, the algorithm calculates the "edit distance" between the non-overlapping child elements of these pairs using the Levenshtein distance [82] that is commonly used for computing string similarity [42].

Stage-to-Stage Matching The matching process extends beyond just the instruction level and is applied to stages as well (lines 7-8). In the first round of stage matching, the algorithm pairs stages based on identical tree representations, using the same logic as for instruction matching. As the algorithm advances to the second round, the criteria are relaxed. Instead of requiring an exact structural match, the algorithm now seeks stages that have closely related instructions. The `computeScore` mechanism calculates stage similarity by evaluating the number of instructions they share, as determined in the first two rounds of instruction matching. This score reflects how closely the stages resemble each other based on their matched instructions. The refinement phase, which calculates the distance between uncommon child elements, is not employed for stages. This is due to the fact that stage matching is inherently dependent on the accuracy of the instruction matching process.

Because the instruction similarities have already been calculated and refined, adding a distance-based refinement for stages becomes redundant.

Matching Preprocessing When matching components, we apply two pre-processing techniques on the input trees (line 2), namely *Shell-Stripping* and *Unification* to deal with specific changes in the Dockerfile when applying some refactorings.

Shell-Stripping: We opt for a simplified representation of shell scripts embedded within the RUN instruction, treating them as a collection of command nodes rather than constructing a separate shell AST. This approach aligns with our primary focus on Dockerfile refactorings, sidestepping the detailed complexities of shell scripting. Moreover, this strategy helps us circumvent the significant computational overhead that a more granular shell script representation could entail. Furthermore, when refactoring operations are performed on the RUN instruction, it's not uncommon for developers to simultaneously update the commands within it. For instance, this might involve adding flags to clear cache post-installation or to ensure non-interactive installations [83]. While these flags are crucial for the operational aspect of the Dockerfile, they can introduce noise in the matching process. To address this, our algorithm incorporates a level of abstraction where flags present in RUN instruction commands are omitted.

Unification: Some Dockerfile refactorings might involve replacing explicit values with ARG or ENV variables, and vice versa. For example, when standardizing configurations across multiple Dockerfiles, explicit values might be replaced with ARG or ENV variables. These variables allow for parameterized builds and run-time configurations. However, the variable names might differ significantly from their original explicit values, leading to discrepancies in the Dockerfile before and after refactoring. Unification addresses this by replacing variable names in the refactored Dockerfile with their corresponding default values (if present). Moreover, in cases where a file is renamed within the same commit, our

algorithm adjusts the filename consistently across the ADD and COPY instructions. This is done by referencing the build context to ascertain the correct path.

Component Matching Example: Figure 5.3 presents examples of component matching between two Dockerfile revisions. Leveraging Shell-striping technique, the commands denoted by ⑧ and Ⓗ were simplified to `RUN pip install requirements.txt`. This modification removes all flags, thus allowing for identical match between the instructions. Through Unification, the instruction marked by Ⓜ was refined. The `APP_MODE` variable was replaced with its default value, as indicated by Ⓚ. Consequently, the command inside the RUN was updated to `RUN /app/backend/scripts/ setup.production.sh`. This ensures it matches exactly with the command represented by ⑬.

One of the key advantages of our matching approach lies in its inherent ability to bypass the constraints of a predefined similarity threshold. Empirical research has demonstrated that the components of Dockerfiles, whether at the stage-to-stage or instruction-to-instruction level, frequently undergo complex modifications. Although not strictly considered refactoring, these changes can occasionally overlap with refactoring operations [14, 3]. These modifications can make the same component appear textually different in both revisions. Trying to set a one-size-fits-all similarity threshold is difficult and often unhelpful due to these varied changes. Our methodology, in contrast, sidesteps this challenge. Instead of relying on mere textual overlaps, it digs deeper, ensuring that components are matched based on their inherent structure and content.

5.1.3 Refactoring Detection

In this subsection, we present our rules for the detection of the 12 refactoring types present in Table 5.1.

Notations: For each revision r of Dockerfile d, we extract the following information:

Table 5.1: Refactoring detection rules

Dockerfile Components	Refactoring Type (defined at [3])	Rule
Instruction	*Add ENV Variable*	$\exists i \in I_d^+ \mid i.t = \text{'ENV'}$
	Add ARG Instruction	$\exists i \in I_d^+ \mid i.t = \text{'ARG'}$
	Replace ADD with COPY	$\exists (s_1,s_2) \in S_d^= , (i_1,i_2) \in I_d^= \mid i_1 \in s_1.I \wedge i_2 \in s_2.I \wedge i_1.t = \text{'ADD'} \wedge i_2.t = \text{'COPY'}$
	Update Base Image	$\exists (s_1,s_2) \in S_d^= \mid i_1 \in s_1.I \wedge i_2 \in s_2.I \wedge i_1.t = \text{'FROM'} \wedge i_2.t = \text{'FROM'} \wedge i_1.image \neq i_2.image$
	Update Base Image tag	$\exists (i_1,i_2) \in I_d^= \mid i_1.t = \text{'FROM'} \wedge i_2.t = \text{'FROM'} \wedge i_1.image = i_2.image \wedge i_1.tag \neq i_2.tag$
	Rename Image	$\exists (i_1,i_2) \in I_d^= \mid i_1.t = \text{'FROM'} \wedge i_2.t = \text{'FROM'} \wedge i_1.image = i_2.image \wedge i_1.alias \neq i_2.alias$
	Inline Run Instructions	$\exists (s_1,s_2) \in S_d^= , (i_1,i_2) \in I_d^=, i_3 \in I_d^- \mid i_1 \in s_1.I \wedge i_2 \in s_2.I \wedge i_3 \in s_1.I \wedge i_1.t = \text{'RUN'} \wedge i_2.t = \text{'RUN'} \wedge i_3.t = \text{'RUN'} \wedge i_2.commands \cap i_3.commands \neq \emptyset \wedge i_1.commands \cap i_3.commands \neq i_2.commands \cap i_3.commands$
	Extract RUN Instructions	$\exists (s_1,s_2) \in S_d^= , i_1 \in I_d^-, i_2 \in I_d^+ \mid i_1 \in s_1.I \wedge i_2 \in s_2.I \wedge i_1.t = \text{'RUN'} \wedge i_2.t = \text{'RUN'} \wedge i_1.commands \cap i_2.script$
	Sort Instructions	$\exists (s_1,s_2) \in S_d^= , (i_1,i_2), (i_3,i_4) \in I_d^= \mid i_1 \in s_1.I \wedge i_2 \in s_1.I \wedge i_3 \in s_2.I \wedge i_4 \in s_2.I \wedge i_1.o < i_2.o \wedge i_3.o > i_4.o$
Stage	*Extract Stage*	$\exists (s_1,s_2) \in S_d^= , s_3 \in S_d^+ \mid \text{dependency}(s_2,s_3)$
	Inline Stage	$\exists (s_1,s_2) \in S_d^= , s_3 \in S_d^- \mid \text{dependency}(s_1,s_3)$
	Move Stage	$\exists s_1 \in S_d^- , s_2 \in S_{d'}^+ \mid \text{match}(s_1,s_2)$

d represents the Dockerfile currently being analyzed.
d' represents any Dockerfile (except d) that was edited in the same commit.
i.commands: Set grouping all command parameters of a RUN instruction i.
dependency(s_1,s_2): Returns True if there is a dependency between s_1 and s_2.
match(s_1,s_2): Returns True if we can match s_1 to s_2.

- S_{rd}: Denotes Dockerfile stages set, with each stage s expressed as (n,I), where n is the stage name and I is the set of instructions for that stage.

- I_{rd}: Denotes Dockerfile instructions, with each instruction i represented as (s,o,t,p), where s is the stage, o is the instruction order within the stage, t is the instruction type, and p are the instruction's parameters, varying by type.

The detection of refactorings occurs after matching the E-ASTs between two revisions r_1 and r_2 of the Dockerfile d using Algorithm 5.1, with I_d^-, I_d^+, $I_d^=$, S_d^-, S_d^+, $S_d^=$ = `component_matching` $(T1,T2)$. This process identifies added, deleted, and modified

components, storing these as $S_d^{+/-/=}$ and $I_d^{+/-/=}$, respectively. The $+$ denotes additions, $-$ denotes deletions, and $=$ indicates matched components.

Examination Order of Refactoring Types: We detect the refactoring types in the order they appear in Table 5.1 by applying the rules shown in the third column of the table. Even though our refactoring detection rules are independent of one another, the order of examination is very important for the accuracy of our approach. We categorize the Dockerfile refactorings based on how they affect the placement of instructions. We commence with refactorings that do not involve instruction relocation and often involve minor modifications or additions (e.g., *Replace COPY with ADD*, *Add ENV Variable*), subsequently advancing to refactorings that shift instructions within the same group (e.g., *Inline Run Instruction*), transitioning next to refactorings that introduce new stages by segmenting existing instructions (e.g., *Extract Stage*), and culminating with refactorings that move instructions to new stages into another Dockerfile (e.g., *Move Stage*). The underlying logic for this examination order is based on empirical research, that shows that localised refactorings are more common than global ones [3]. This implies that any added or removed instructions are more likely a result of specific, localized changes rather than broad alterations. Using this ordering, as we navigate through each detected refactoring, we systematically filter out the associated instructions/stages from the pool of potential additions or deletions, classifying them in their respective matched sets. This simplified approach reduces the number of instructions assessed in subsequent refactoring categories, lowering the possibility of misinterpretation and increasing the detection accuracy. For example, When we detect an *Inline Run Instruction*, we make sure it isn't mistakenly identified as part of a more complex refactoring like *Extract RUN Instruction*. Furthermore, by accurately identifying an *Inline Stage*, we prevent it from being misclassified as potential sources for *Move Stage* refactoring. This distinction is crucial given the complexity of this refactoring as it requires a thorough examination across all pre-existing Dockerfiles within the same

commit to accurately identify and relocate the desired stage. We note that, If, in the process of identifying a *Move Stage*, a stage in the modified Dockerfile cannot find a match within the sets of unmatched stages from all Dockerfiles in the same commit, we then generate the for the newly created Dockerfiles.

Context Aware Refactoring Detection: Our matching algorithm is designed with a focus on one-to-one component matching by always selecting the best match for a specific instruction or stage, a choice to sidestep the excessive computational demand and the potential for false positives associated with one-to-many or many-to-one matching scenarios. This complexity arises from the need to evaluate a significantly larger set of potential matches. Yet, it is imperative to acknowledge that our system does not natively accommodate the intricacies associated with a subset of refactoring types. Specifically, only three out of the twelve refactoring types we analyze-*Inline Run Instruction*, *Extract Stage*, and *Inline Stage*-may necessitate such complex matching of one-to-many or many-to-one due to their inherent nature of merging or splitting processes.

To address these cases, we have implemented a two-faced strategy. Firstly, we have adapted the detection rules for these three refactorings to consider unmatched elements alongside matched ones. This allows us to identify signs of more intricate refactoring activities that may not be immediately apparent through direct matching. For instance, if a RUN instruction or stage remains unmatched, it could indicate a refactoring event such as inlining or extraction. By analyzing these unmatched components, we can infer the occurrence of such complex refactorings indirectly.

Secondly, we enhance the algorithm's contextual awareness by updating the matching sets as new refactorings are detected. This dynamic updating process ensures that each subsequent rule is applied with the most current understanding of the code's structure. For example, after detecting an *Inline Run Instruction*, the matching set is updated to reflect

this transformation (many-to-one), which is essential for accurately identifying subsequent refactorings like *Sort Instructions* that rely on the updated context.

5.2 Current Results for DRMiner Evaluation

Table 5.2: Precision, recall, and F1 per refactoring type

Refactoring Type	**RQ1**						**RQ2**					
	TP	FP	FN	Precision	Recall	F1	TP	FP	FN	Precision	Recall	F1
Move Stage	1	0	0	1 100%	1 100%	1.00	1	0	0	1 100%	1 100%	1.00
Inline Stage	2	0	0	1 100%	1 100%	1.00	20	0	0	1 100%	1 100%	1.00
Update Base Image	80	8	0	0.91 91%	1 100%	0.95	63	58	0	0.52 52%	1 100%	0.68
Inline Run Instruction	31	0	1	1 100%	0.97 97%	0.98	121	0	25	1 100%	0.83 83%	0.91
Extract Run Instruction	37	0	2	1 100%	0.95 95%	0.97	12	0	0	1 100%	1 100%	1.00
Add ARG Instruction	30	1	0	0.97 97%	1 100%	0.98	105	3	0	0.97 97%	1 100%	0.99
Add ENV Variable	74	3	0	0.96 96%	1 100%	0.98	195	0	0	1 100%	1 100%	1.00
Sort Instructions	72	3	0	0.96 96%	1 100%	0.98	104	7	0	0.94 94%	1 100%	0.97
Rename Image	5	0	8	1 100%	0.38 38%	0.56	21	0	0	1 100%	1 100%	1.00
Extract Stage	6	0	0	1 100%	1 100%	1.00	15	0	0	1 100%	1 100%	1.00
Update Base Image Tag	15	1	0	0.94 94%	1 100%	0.97	94	1	0	0.99 99%	1 100%	0.99
Replace ADD with COPY	N/A	N/A	N/A	N/A	N/A	N/A	29	0	0	1 100%	1 100%	1.00
Overall	353	16	11	0.951 95.1%	0.97 97%	0.96	780	69	25	0.919 91.9%	0.969 96.9%	0.941

We implemented our approach as a self-contained tool, DRMiner, designed to identify and analyze Dockerfile refactorings. It is engineered to accept a GitHub repository URL as input. Upon execution, DRMiner systematically scans the specified project for Dockerfile changes, offering users the flexibility to target specific Dockerfiles or commits for a more focused analysis. Alternatively, users can opt for a broader evaluation by conducting an exhaustive search for refactorings across all Dockerfiles in the given repository. The output is a report that encapsulates the refactoring activities detected within the project's Dockerfile history.

5.2.1 Research Questions

To evaluate DRMiner, we answer three main research questions (RQs). The first two focus on ***quantitative analysis*** where we use **precision**, **recall** and **F1 score** as evaluation metrics, while the last one is dedicated to ***qualitative analysis***.

RQ1: To what extend does DRMiner correctly implement the refactoring definitions established by prior empirical study?

The effectiveness of DRMiner relies on its adherence to the refactoring definitions established by ksontini et al. [3]. This study, which was the first to define Dockerfile refactorings, serves as the foundation for creating DRMiner's rule set. We ensure DRMiner's outputs are aligned with empirically derived definitions of refactoring by benchmarking against their dataset.

To answer **RQ1**, we employed a dataset of refactoring instances that was made publicly available by ksontini et al. [3]. Three lab members of this empirical study conducted a thorough manual validation of each refactoring instance within the dataset. For the purposes of our study, we have updated the dataset by excluding commits and projects that are no longer accessible, due to repositories being deleted or potentially converted to private access. Following this update, the dataset includes instances from 310 Dockerfiles distributed across 73 commits from 35 open-source projects.

RQ2: What is the effectiveness of DRMiner in generalizing its refactoring detection across various Dockerfile commit history?

The practical value of DRMiner extends beyond mere correctness; it must effectively generalize across varied codebases. This research question aims to assess DRMiner's adaptability to the diverse styles and practices present in Dockerfile commit histories.

To answer **RQ2**, we utilized the Google BigQuery GitHub Public Dataset [2], released in 2016. This dataset provides access to the source code of over 2 billion files from 3 million open-source repositories. We used the most recent snapshot available at the start of this project (May 25, 2023) to collect commit messages containing the pattern `"refactor.* Dockerfile"`. This pattern was chosen to maximize the likelihood of capturing commits specifically related to Dockerfile refactorings. Our query returned a result set of 738 commits across 577 projects. To refine our selection process for RQ2, we employed GitHub star counts as a criterion, leveraging the insight that popular and young projects generally exhibit fewer Dockerfile quality issues, as suggested by Wu et al.'s exploration of Dockerfile practices [1]. This approach allowed us to focus on a representative sample of projects, capturing a diverse range of refactoring scenarios. Specifically, we selected the 100 most popular GitHub projects, along with the 100 least popular. We further refined the dataset by excluding projects previously studied by Ksontini et al. [3] and discarding forks, resulting in 351 Dockerfiles from 223 commits across 187 unique projects. These projects represent a spectrum of the open-source ecosystem, including both renowned and emerging projects from varied domains, with owners ranging from tech giants like Google, Mozilla and dotNET to smaller, specialized entities. The demographics of studied project can be found in our replication package [84].

We conducted a manual validation study with four active industry developers with an average of three years of experience in Docker development, who volunteered to participate in our study as part of an industry-sponsored research collaboration to annotate the Docker refactorings. We followed the Kappa cross-validation process to annotate the refactorings within the dataset. Each filled out a pre-study survey that collected background information, such as their programming experience and their role within their companies. To improve the survey outcome, we have made every possible effort to avoid any potential bias. We

[2]https://codelabs.developers.google.com/codelabs/bigquery-github

gave participants a two-hour lecture about Docker quality assessment and refactoring. We provided general knowledge regarding refactoring and showed them how to read and interpret the refactoring practices and focused on explaining the required steps. During the validation process, we assigned on average 111 commits per participant to ensure that each commit was evaluated by two developers and we considered it relevant if both of them agreed (The overall Cohen's kappa was 0.96). The details of the participants and the projects they evaluated are found in our replication package [84].

RQ3: Why do DRMiner misdetect some refactorings?

In **RQ2**, even though the tool achieved commendable performance, a percentage of refactorings were misdetected. Certain refactorings were incorrectly identified as refactorings, which we refer to as false positives. In addition, certain actual refactorings were overlooked, which we refer to them as false negatives.In this research question, we want to understand the reasons behind these false positives and negatives. The answer to this question may help us to discover further hidden factors that are related to the characteristics of Dockerfile refactoring practices and to improve our current detection algorithm.

To answer **RQ3**, we performed a qualitative analysis of false positives and false negatives to search for the main causes of the wrong misdetections.

5.2.2 Results

Table 5.2 illustrates the precision, recall, and F1 scores for DRMiner across detected refactorings for both RQ1 and RQ2.

In response to RQ1, DRMiner detection rules demonstrated strong alignment with empirically established refactoring definitions, by achieving a high overall precision of 95.1% and recall of 97% .

The evaluation showed that DRMiner achieved 100% precision and recall in 4 refactoring categories: *Extract Run Instruction*, *Extract Stage*, *Inline Stage*, and *Move Stage*. This

outcome indicates a complete alignment of the tool's detection rules with the empirical definitions in these categories. In six other refactoring categories - *Add ENV Variable*, *Sort Instructions*, *Update Base Image*, *Update Base Image Tag*, and *Add ARG Instruction* - DRMiner maintained high precision (ranging from 91% to 97%) alongside a perfect recall of 100%. However, a notable deviation was observed in the *Rename Image* category, where, despite a high precision of 100%, the recall stood at 38%. In this category, DRMiner accurately identified 5 instances but overlooked 8 others. Further investigation into this deviation revealed a specific pattern: all 8 misclassified instances occurred within the same project [3]. These instances involved changes in the base image names rather than the aliases. The refactoring rules, as defined, focus on changes in image aliases, which led to these 8 instances being incorrectly classified as *Update Base Image* refactorings. Overall, DRMiner's performance was notable, achieving an average precision of 95.1% and recall of 97%, culminating in an F1 score of 0.96 across all refactoring types. It should be noted that the dataset provided by ksontini et al. [3] presented only one occurrence of the *Replace ADD with COPY* refactoring and the repository containing this instance is no longer available on Github, thus we were not able to compute its precision and recall (i.e., the rows in Table 5.2 with N/A).

In response to RQ2, DRMiner adeptly generalized its refactoring detection across diverse Dockerfile contexts, correctly identifying 780 out of 805 refactoring operations, achieving an average precision of 91.9% and recall of 96.9%. DRMiner achieved 100% precision and recall in seven refactoring types, including *Inline Stage*, *Extract Stage*, *Extract Run Instruction*, *Replace ADD with COPY*, *Rename Image*, *Add ENV Variable* and *Move Stage*. This indicates its strong capability in consistently detecting these refactorings across a wide range of Dockerfile contexts. The tool also performed well in *Sort Instructions*, *Add ARG Instruction*, and *Update Base Image Tag* categories, maintaining high precision (94%

[3] https://github.com/uselagoon/lagoon/commit/fd6b0db52f09dc5

to 99%) along with complete recall. However, for *Inline Run Instruction*, while DRMiner achieved 100% precision, the recall was lower at 83%, resulting in an F1 score of 0.91. This was attributed to the tool missing 25 instances of this refactoring type. Moreover, for *Update Base Image* category, despite achieving a perfect recall of 100%, the precision was notably lower at 52%. This outcome indicates that the tool, while successfully capturing every actual *Update Base Image* refactoring, also tended to extend its classification to additional changes, resulting in 58 instances being incorrectly identified as refactorings in this category. Overall, DRMiner's performance was notable, achieving an average precision of 91.9% and recall of 96.9%, culminating in an F1 score of 0.941 across all refactoring types. The dataset used for our analysis encompassed 351 Dockerfiles . Upon thorough manual validation, it was determined that 292 of these Dockerfiles contained refactoring operations, accounting for a total of 805 individual refactoring operations. DRMiner demonstrated its effectiveness by correctly identifying 780 of these operations.

In response to RQ3, DRMiner's primary challenges were accurately discerning complex RUN command mergers without the && operator and understanding context-dependent changes in base images.

Regarding the *Inline Run Instruction* refactoring, which accounted for all 25 false negatives, DRMiner failure in detecting this category arise primarily when the inlining process is accompanied by command modifications within a RUN instruction. The most common occurrence, representing 56% (14/25) of these cases, involved merging multiple package installation commands into a single command. This merging often excluded the use of the && operator. An illustrative case[4] showcased the transition from individual `RUN dnf install <package1>` and `RUN dnf install <package2>` commands to a unified format `RUN dnf install <package1> <package2>`. Additionally, 36% (19/25) of false negatives were attributed to instances where the entire structure of a command was modified,

[4]https://github.com/twtiger/gosecco/commit/7b516d243360e5f

yet its functional behavior remained unchanged. This scenario requires a deeper semantic analysis of the command's purpose and its contextual use within the Dockerfile. For example, in one instance[5], the developers transitioned from commands like `RUN su - webapp -c gem install bundler` to `RUN /home/webapp/.rbenv/shims/gem install bundler` following a shift in the user context to `USER webapp`. This alteration in execution context and command structure led to DRMiner missing 9 out of 22 *Inline Run Instruction*. The remaining 8% (2/25) of cases were marked by the removal of specific elements from commands, such as certain packages or parts of a multi-step process. A notable example[6] involved updating the base image to `nginx`, leading to the removal of the `nginx-core` package from the installation command.

In the *Update Base Image* refactoring category, DRMiner recorded 56 false positives, predominantly occurring when developers added new repositories to their base images, representing 91.4% (53/58) of these cases. This addition typically happens after developers push their custom-built Docker images to Docker Hub or a private registry. They then update the FROM instruction in the Dockerfile to pull the image from these new locations. DRMiner misinterpreted these changes as updates to the base image, rather than recognizing them as shifts in the image's source repository. An example can be seen in this case[7]. This underlines the need for more contextual information, such as commit messages or build scripts, to accurately interpret these changes and distinguish them from genuine base image updates. The remaining 8.6% (5/58) of false positives were related to the use of template placeholders in the base image. These unresolved placeholders during static analysis led to misinterpretations, as seen in the first Dockerfile of this commit[8].

[5]https://github.com/ipepe/pnpr/commit/4c46e275e48ed
[6]https://github.com/lmenezes/elasticsearch-kopf/commit/0572b384146a
[7]https://github.com/BBVA/openstack-k8s/commit/d84a4e3c65213d
[8]https://github.com/dotnet/dotnet-docker/commit/5a12b509a

In the context of the *Sort Instructions* refactoring, DRMiner's false positives, amounting to 7 cases, were primarily due to instances of instruction reordering that did not affect directly to Docker build optimization. A prevalent cause of these false positives was the rearrangement of ARG and ENV instructions. Unlike RUN or COPY instructions, changes in the order of ARG and ENV do not impact the creation of layers in the Docker image, as they are used for setting environment variables or passing arguments to the build process without altering the image's layer structure. An example of the repositioning of these instructions is seen in the first Dockerfile of this commit[9]. Additionally, some of the false positives were linked to developers making corrective modifications in their Dockerfiles. An example of such correction is the repositioning of the WORKDIR instruction. While these adjustments can be essential for the correct functioning of the Dockerfile, they do not necessarily align with the typical rationale behind *Sort Instructions* refactoring.

For the *Add ARG Instruction* refactoring category, the misdetections (3 instances) primarily occurred when developers substituted ENV variables with ARG, as seen in this example[10]. This practice is common when enhancing Dockerfile flexibility, as ARG allows for dynamic parameterization during the build phase, unlike the more static ENV.

5.3 Threats To Validity

We now discuss the threats to the validity of our study following the guidelines for case study research [85].

Threats to internal validity are factors that may influence our independent variables and that were not taken into account. In our study, we developed rules based on docker refactorings defined by Ksontini et al. [3]. This process poses a threat to internal validity, as creating and implementing these rules might introduce biases or inaccuracies. To counter this, we have established the **RQ1** that focuses on validating the DRMiner against the dataset

[9]https://github.com/eleidan/docker-rust/commit/11cd48fa727ead
[10]https://github.com/dtzar/helm-kubectl/commit/e08f9493d3e9d64c

from which these refactorings were originally derived. This step is crucial for ensuring that DRMiner accurately reflects the intended refactorings and provides reliable results in line with the theoretical framework of Ksontini et al. [3].

Threats to construct validity are concerned with the relationship between the treatments and the outcome. In our research, We paid attention to the complexity in representing structural and semantic shifts in Dockerfiles, which may be missed using traditional Dockerfile AST techniques. To counter this, we developed a specialized grammar and introduced the Enhanced Abstract Syntax Tree (E-AST). This tailored approach captures the intricacies of Dockerfile structure, significantly enhancing the precision and reliability of our detection methods.

Threats to external validity affect the generalizability of our results. We carefully ensured that the datasets for evaluating tool correctness **RQ1** and extensibility **RQ2** did not overlap. Additionally, to mitigate potential biases inherent to any specific dataset, we diversified our dataset in **RQ2**, selecting projects based on GitHub star counts. This included a mix of both highly starred and less popular projects, ensuring a wide range of Dockerfile refactoring patterns were represented.

5.3 Conclusion

This chapter presents DRMiner, a tool designed for Dockerfile refactoring detection and analysis. DRMiner fills a critical void in containerization by offering substantial enhancements in Dockerfile maintenance. It achieves this through its E-AST based component-matching algorithm and comprehensive detection rules. Our evaluation demonstrates DRMiner's exceptional proficiency, with the tool achieving a high precision of 95.1% and recall of 97% in alignment with established refactoring definitions (RQ1). Furthermore, DRMiner exhibits remarkable generalization capabilities across varied Dockerfile contexts, successfully identifying 780 out of 805 refactorings, with an average

precision of 91.9% and recall of 96.9% (RQ2). These results underscore DRMiner's significance in the rapidly evolving Docker landscape, contributing to improved reliability, performance, and error management in containerized applications, as well as facilitating a deeper understanding of refactoring practices.

CHAPTER SIX

A METHOD FOR SPECIFICATION AND DETECTION OF ANTI-PATTERNS

6.0 Introduction

Creating Docker images is inherently iterative, often involving multiple revisions of Dockerfiles—the scripts that automate the image-building process. While this flexibility has driven the rapid adoption of Docker, it has also led to the proliferation of Dockerfile anti-patterns. These anti-patterns represent recurring ineffective practices that can degrade Docker images' quality, maintainability, and efficiency. Unlike localized code smells, which focus on specific coding issues, anti-patterns affect the broader architecture and design of the Docker image, leading to increased technical debt, bloated images, and complex maintenance challenges.

Addressing these anti-patterns is not merely a corrective measure but a strategic necessity for ensuring the long-term viability of Docker-based systems. Refactoring, the disciplined process of restructuring code to improve its internal structure without altering its external behavior, plays an important role in this context. When applied to Dockerfiles, refactoring serves as a critical mechanism for counteracting the negative impacts of anti-patterns, thereby enhancing the maintainability, efficiency, and scalability of Docker images [86]. However, successful refactoring hinges on a deep and precise understanding of the underlying anti-patterns—where they occur, how they manifest, and the specific ways in which they compromise the Dockerfile's architecture. Without this foundational insight, refactoring efforts may be misdirected or incomplete, failing to address the root causes of technical debt and architectural degradation.

Despite the critical impact of Dockerfile anti-patterns, much of the existing research has concentrated on identifying Dockerfile smells—issues related to coding practices within

the Dockerfiles [31, 5, 32], such as improper use of bash scripts. While identifying and addressing these smells is essential, it does not fully address the more significant design flaws that anti-patterns represent.

This chapter addresses this gap by introducing a novel approach for specifying and automatically detecting Dockerfile anti-patterns. We begin by conducting a comprehensive domain analysis to identify common anti-patterns that frequently occur in industry practice. Through this analysis, we define five new Dockerfile anti-patterns that capture critical design flaws in Docker image construction. These anti-patterns are formalized using a set of 12 metrics, which provide a quantitative basis for their detection through static analysis. A key contribution of this work is the development of a tool that automates the detection of these anti-patterns.

6.1 Approach for Specifying and Detecting Anti-patterns in Docker-projects

The main stages for developing the specification and detection approach for Dockerfile anti-patterns are illustrated in Figure 6.1 and can be summarized as follow:

Step 1. Domain Analysis:: Key concepts are identified in the text-based descriptions of the common mistakes, smells, and bad practices made by developers when creating, designing, and configuring Docker images present in the literature and some online materials. These key concepts will form a unified vocabulary of reusable concepts to describe the Dockerfile anti-patterns.

Step 2. Formal Specification:: The ideas that form a vocabulary are consistently and formally integrated to specify anti-patterns.

Step 3. Implementation of Detection: The specifications are converted into algorithms that may be used to detect Dockerfile anti-patterns.

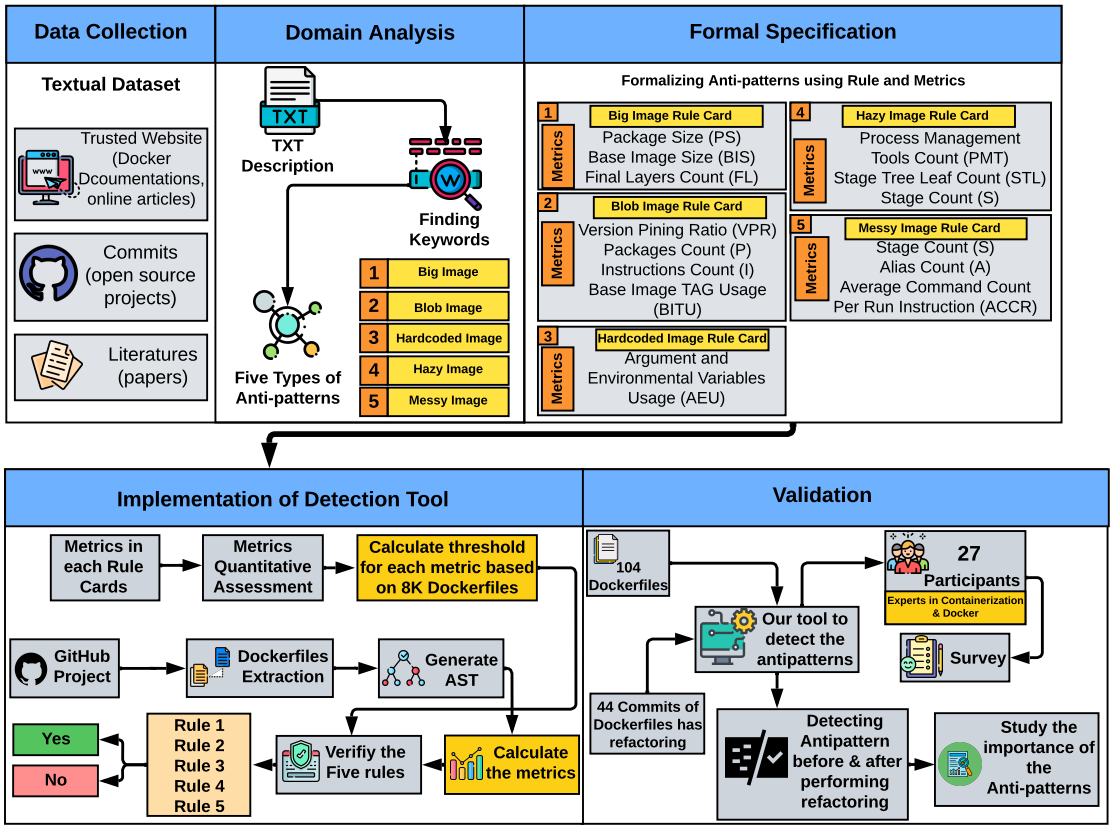


Figure 6.1: Approach overview

Step 4. Evaluation:Conduct a survey with practitioners to assess the effectiveness of our tool in finding anti-patterns in Docker projects, in addition to studying the importance of these anti-patterns from the perspective of developers.

The following subsections describe the three steps of our approach using a common pattern: input, output, and description. The ***input*** is the data or information that is required to perform the desired step, the ***output*** of our approach is the result that is produced after processing the input, and the ***description*** is a concise explanation of what the step does, It provides an overview of the method used. Some steps are illustrated by an example if needed.

6.1.1 Domain Analysis

We perform at the first stage the domain analysis, which "is the process in which information that describes the software systems is identified, recorded, and structured to make the software reusable for future support"[87]. At this stage, we mine examples of inconsistencies in real open-source projects, and we study the Dockerfile common mistakes described in online materials and articles. This allows for uncovering and specifying important characteristics of Dockerfile anti-patterns and, hence detecting the issues related to Dockerfile, such as poorly designed static features linked to the Dockerfile content. Specifying these features allows for defining a vocabulary of a set of rules to express Dockerfile anti-patterns. This will provide software developers with high-level domain-specific abstractions to recognize various anti-pattern qualities.

6.1.1.1 Process **Input:** Text-based descriptions of common mistakes and bad practices made by developers when creating, designing, and configuring Docker images present in the literature and trusted online materials.

Output: A textual list of the key concepts which form a vocabulary for Dockerfile anti-patterns.

Description: We create a set of core concepts and fundamental ideas that will be utilized to identify and describe new anti-patterns. The *first step* was to manually collect and inspect textual descriptions of Dockerfile smells, bad and best practices, including typical mistakes, and samples of real technical debts and defects. We looked at the most critical defects that happened in current Dockerfile projects that were present in the refactoring commits given by Ksontini et al. [86]. We also investigated frequent Dockerfile errors, as well as bad and good practices, stated in literature and various online materials [5, 88, 29, 89].

The *second step* was to determine the most important ideas to build a standardized vocabulary of reusable ideas and classify potential anti-patterns. This vocabulary was

created with the help of a set of measurable attributes that represent notions represented through measurements of internal qualities of Dockerfile elements (instructions, layers, and stages, etc.) as well as structural attributes (multi-staging, stage dependencies, etc.).

The *third step* involved manually organizing all anti-patterns with respect to one another and creating a taxonomy that displays all anti-patterns on a single map. Using a set of operators such as intersection and union, the map organizes and mixes patterns and other related essential notions.

Table 6.1: Key-concept examples extracted from Docker Documentation

How to keep your images small! Small images are faster to pull over the network and faster to load into memory when starting containers or services. There are a few rules of thumb to keep the image size small: Start with an ***appropriate base image***. For instance, if you need a JDK, consider basing your image on the official OpenJDK image, rather than ***starting with a generic ubuntu image and installing OpenJDK as part of the Dockerfile.*** Use ***multistage builds***. For instance, you can use the maven image to build your Java application, then reset to the tomcat image and copy the Java artifacts into the correct location to deploy your app, all in the same Dockerfile. This means that your final image does not include all of the libraries and dependencies pulled in by the build but only the artifacts and the environment needed to run them. If you need to use a version of Docker that does not include multistage builds, try to ***reduce the number of layers in your image*** by minimizing the number of separate RUN commands in your Dockerfile.

Process Implementation Example: Containers, in contrast to virtual machines, are designed to be lightweight and take up less space (container images are typically tens of MBs in size). As a result, it is highly recommended to constantly aim for smaller images

when creating Docker containers. Images with a reduced size are easier to transfer and deploy.

We provide an example of a text-based description in Table 6.1, extracted from the official guidelines of Docker documentation on how to avoid heavy images. From this description, we extracted the following key concepts (in italic in the table): *Utilizing unnecessary heavy generic images (like ubuntu) and downloading heavy dependencies and packages on top of it (like JDK).* This should be replaced with a lighter official base image (like OpenJDK) and thus prevent downloading packages. *Multi-stage building* and dividing Dockerfile into various stages to transmit the needed item from one stage to another and finally provide the completed artifact in the last step. In this manner, the final image won't have any unnecessary elements. *A Docker image takes up more space with every layer you add to it.* Therefore, the more layers you have, the more space the image requires.

From this textual description, we inferred the vocabulary of this anti-pattern: its measurable properties include the concepts of Base Image Size and installed Dependencies size, and it is structural features include the concept of the multi-stage building. Measurable properties are described by values expressed using keywords such as high or low. These attributes may be coupled using set operators such as intersection and union. More details on the properties and their possible values for the key concepts are given in subsubsection 6.1.2.2, where we show the formal BNF rule of the Big Image and other anti-patterns generated from the domain analysis, its grammar, as well as the list of the attributes and values.

6.1.1.2 Anti-patterns Vocabulary Besides, the Big Image anti-pattern defined as follows:

Big Image: Represents an image that embraces practices that could affect the final image size, leading to large images with a very slow building and deployment.

The domain analysis step yielded a total of four additional anti-patterns, namely:

Blob Image: Treating containers as virtual machines capable of running several processes at the same time is a typical mistake [90]. While a container may be used in this manner, most of the benefits of the container paradigm are lost. Take a standard MySQL/PHP stack, for example, we may be tempted to run all of the components in a single container. However, utilizing two or three different containers is the recommended approach as stated in Docker Documentation "each container should do one thing and do it well" [91]. On the other side, including a CI/CD pipeline inside a Docker image is a typical mistake since developers often mix commands that must execute in the Docker container with instructions that must run in a CI build process [89]. Docker isn't a continuous integration solution in and of itself. Container technology may be employed in a CI/CD pipeline, which is a whole distinct scenario. Blob Image corresponds to an image with the lion's share of responsibilities (includes CI/CD pipelines or includes several apps), making it difficult to build and maintain.

Hardcoded Image: When only the configuration changes, it is a typical practice to hardcode it and build a new Dockerfile or alter an old one. A hardcoded Image corresponds to an image that uses a hardcoded configuration and contains duplicated fragments, which makes it challenging to maintain. Rather than hard-coding information, ENV variables can be used to specify regularly used fragments, such as version numbers, making the image simpler to modify. Following the pattern of variables in a program, this method allows us to edit a single ENV instruction to automatically update the software version in your container. Furthermore, ARG variables may be used in conjunction with ENV variables to enable setting variables at build time.

Hazy Image: A Docker image is a blueprint for Docker containers that includes the application as well as everything else you will need to execute it [90]. Dockerfiles that conduct "unexplainable" or "bug-prone" steps should be avoided. Because the OS, library versions, settings, directories, and applications are all enclosed within the container, it is immutable. We should ensure that the same image that was created will appear in every

setting and behave in the same way we expect it. Thus, it is important to tag the base image (not to use the latest TAG) and specify the version for every downloaded dependency to prevent any unexpected bugs in the future. A hazy Image represents an image that embraces bug-prone practices or baffling steps, making it difficult to maintain or reuse.

Messy Image: Dockerfiles have a basic, clean, and straightforward syntax, making them very simple to write and utilize. They're meant to be self-explanatory, particularly because they allow for a clear description of the different steps, much like a well-written program. A messy Image represents an image that is difficult to read and understand.

6.1.2 Formal Specification

In this subsection, we formally specify the five anti-patterns we described earlier. For each anti-pattern, we provide the rule card and the list of metrics required to define it.

6.1.2.1 Process **Input:** A vocabulary and taxonomy of anti-patterns.

Output: Specifications detailing the formal rules to apply on a Dockerfile to detect the specified anti-patterns.

Description: We formalize the concepts and properties of anti-patterns to specify detection rules at a high level of abstraction. Using the smell vocabulary and taxonomy, we map rules with smells and rule cards with anti-patterns. Each anti-pattern in the taxonomy corresponds to a rule card. Each smell associated in the taxonomy with an anti-pattern is described as a rule. The properties in the taxonomy are directly used as metrics to quantify the rules.

6.1.2.2 Anti-patterns Rule Cards: We specified anti-patterns using rule cards, which are collections of rules formalized using Backus-Naur Form (BNF) grammar [92]. The keyword RULE_CARD is used to identify a rule card (line 1), which is then followed by a name and a set of rules that describe a unique anti-pattern. A rule (line 3) defines a metric or a mixture of rules using set operators such as intersection, union, and inclusion (line 5). A rule may also relate to a previously established rule card. A metric (line 6) assigns a numerical or ordinal

value to an identifier. A 3-level scale is used to indicate ordinal values: high, medium, and low (line 8). Ordinal values are used to define values relative to all the Dockerfile investigated, while numerical values are utilized to set thresholds using comparators (line 9). We use the box-plot statistical methodology to create ordinal values and connect them to real metric values, which is a common practice [93, 94]. The specifications of the five defined anti-patterns were as follow:

Big Image Rule Card: The rule card of the Big Image is presented in Figure 6.2, which is characterized by the union and intersection of three rules (line 2). An image is a Big Image if it has both large-sized packages and a Base Image (lines 4 and 5), or if it has an overly layered structure (line 6).

```
1.  Rule_Card:  BigImage {
2.  RULE: BigImage{ UNION OnionImage ( INTER HeavyInstalledPackages
3.   HeavyBaseImage) };
4.  RULE: HeavyInstalledPackages { PACKAGES_SIZE (PS): HIGH };
5.  RULE: HeavyBaseImage{ BASE_IMAGE_SIZE (BIS): HIGH};
6.  RULE: OnionImage {  FINAL_LAYERS_COUNT (FL):  HIGH};
    };
```

Figure 6.2: Rule card of Big Image

Blob Image Rule Card: The rule card of the Blob Image is illustrated in Figure 6.3, which is characterized by the union and intersection of three rules (line 2). An image is a Blob Image if it includes a high number of independent stages (line 3 and line 5) or if it embraces several services/processes (line 4).

```
1.  Rule_Card: BlobImage {
2.  RULE: BlobImage { UNION MultiProcess
    (INTER IndependentStages HighNumberOfStages ) };
3.  RULE: IndependentStages { STAGE_TREE_LEAF_COUNT (STL) > 1 };
4.  RULE: MultiProcess{ PROCESS_MANAGEMENT_TOOLS_COUNT (PMT) > 1};
5.  RULE: HighNumberOfStages { STAGE_COUNT (S): HIGH};
    };
```

Figure 6.3: Rule card of Blob Image

Hardcoded Image Rule Card: The rule card of Hardcoded Image is presented in 6.4, which is characterized by very low usage of both ENV and ARG variables(line 2).

```
1.  Rule_Card: HardcodedImage {
2.  RULE: HardcodedImage { ARG_ENV_USAGE (AEU): LOW };
3.  };
```

Figure 6.4: Rule card of Hardcoded Image

Hazy Image Rule Card: The rule card of the Hazy Image is illustrated in Figure 6.5, which is characterized by the union of four rules (line 2). An image is a Hazy Image if it presents a low version pining (line 3), if it has a small package count (line 4), if it contains a small number of instructions (line 5), or if the base images were untagged (line 6).

```
1.  Rule_Card: HazyImage {
2.  RULE: HazyImage { UNION LOWVersionPining SmallPackagesNumber
    LowNumberOfInstruction UntaggedBaseImages) };
3.  RULE: LOWVersionPining { VERSION_PINING_RATIO (VPR) : LOW };
4.  RULE: SmallPackagesNumber { PACKAGES_COUNT (P) : LOW };
5.  RULE: LowNumberOfInstruction { INSTRUCTION_COUNT (I): LOW};
6.  RULE: UntaggedBaseImages{ BASE_IMAGE_TAG_USAGE (BITU)= 0 };
    };
```

Figure 6.5: Rule card of Hazy Image

Messy Image Rule Card: The rule card of the Messy Image is presented in Figure 6.6, which is characterized by the union of two rules (line 2). An image is a Messy Image if it contains unnamed stages (line 3) or if it has RUN instructions with non-formatted commands (lines 6 and 5).

```
1.  Rule_Card: MessyImage {
2.  RULE: MessyImage{ UNION UnexplainableStages MessyRunInstructions };
3.  RULE: UnexplainableStages { ALIAS_COUNT (A) < STAGE_COUNT (S) -1 };
4.  RULE: PopulatedRunInstructions{ AVG_CMD_COUNT_PER_RUN (ACCR) : MEDIUM};
5.  RULE: MessyCommands { AVG_LINE_COUNT_PER_RUN (ALCR) <
    AVG_CMD_COUNT_PER_RUN (ACCR) };
6.  RULE: MessyRunInstructions{ INTER PopulatedRunInstructions MessyCommands };
    };
```

Figure 6.6: Rule card of Messy Image

6.1.2.3 Metrics:

As aforementioned, to make the concepts of the newly defined anti-patterns vocabulary more tangible and accessible to experimental confirmation, using a compact and carefully

specified set of measurable properties or metrics was necessary. We have developed a list of 12 metrics to specify the five different anti-patterns. These metrics can be listed as follows:

1. **Instructions Count (*I*):** measures the total number of instructions in a single Dockerfile.

2. **Base Image Size (*BIS*):** indicates the base image size in megabytes (MB).

3. **Process Management Tools Count (*PMT*):** When running multiple services inside a single container, Process Management Tools such as Supervisord [1] are generally used to better control, manage, and restart the processes within a single container. *PMT* indicates the number of used Process Management Tools in a single Dockerfile.

4. **Final Layers Count (*FL*):** A Docker image consists of a set of read-only layers. Instructions Layers are created by using only the RUN, COPY, and ADD. other instructions provide temporary intermediate images but do not affect the final build size. It is important to note that in a multi-stage context, only the final stage layers add to the final build size. *FL* indicates the number of final layers within an image.

5. **Packages Count (*P*) and Packages Size (*PS*):** In order to fill the application's requirements, users can set up all the dependencies in the image using RUN instruction commands. In Linux-based images, apt, apk, dpkg, yum, dnf, and rpm are the most commonly used dependency managers for installing the application's environmental requirements [95]. We note that we only supported these dependencies managers in the proposed work. *P* indicates the total number of packages installed within a single Dockerfile and *PS* measures the total installed packages sized in megabytes (MB).

6. **Stage Count (*S*) and Stage Tree Leaf Count (*STL*):** Multi-stage Docker builds are used as a means to create images that derive from several bases or intermediary

[1] http://supervisord.org/

images. We may utilize many base images as well as prior intermediate image layers to create a new image layer using multi-staging. It essentially enables us to create a whole hierarchy of Docker instructions that can be used to generate several sets of images with various functionalities, all from a single Dockerfile.

We generated a tree to represent the dependency and hierarchy of the different intermediate images (stages) within a single Dockerfile. S indicates the number of stages used, while STL represents the number of leaf nodes in the Stage Tree.

7. **Alias Count (A):** Alias is used to describe a symbolic name of an element. In a multi-staging context, we use aliases to identify different stages and to describe it is purpose. A measures the alias count in all stages within a single Dockerfile. We note that in a single-stage context A=1, as using an alias is not mandatory.

8. **Base Image TAG Usage ($BITU$):** Similar to package pining, a Docker image is tied down to a specific TAG that conveys information about the base image versions or variants. If TAG is not specified Docker images will be reliant on the time they were created and will automatically download the latest release in the next build. $BITU$ measures the TAG usage in all base images within a single Dockerfile.

$$BITU = \frac{TBI}{S} \tag{6.1}$$

TBI: the number of tagged base images.

S: stage count.

9. **Argument and Environmental Variables Usage (AEU):** Environmental variables are used by an application to control how containers should run or to set a dynamic configuration. Argument variables, on the other hand, are used to define common identifiers (e.g., software version and file/directories paths) or to specify variables that users can pass at build time to the builder. Both Env and Arg variables can be used

inside other instructions such as COPY, ADD and RUN, leading to a dynamically changing Dockerfile. *AEU* measures the usage of Env and Arg variables among Dockerfile instructions.

$$AEU = \tanh\left(\frac{(IE+IA)}{(A+E)}\right) \times \tanh\left(\frac{(IE+IA)}{I}\right) \tag{6.2}$$

IE: the number of instructions having at least one Env variable.

IA: the number of instructions having at least one Arg variable.

A: the number of Arg variables.

E : the number of Env variables.

I : the total number of instructions where we can use Env or Arg variable. We note that if *A* and *E* are equal to 0, then *AEU* is 0.

10. **Version Pining Ratio (*VPR*) :** The dependencies needed for an application to run in a specific container must be tied down to a certain version in order to get a consistent build. Otherwise, Docker images will be reliant on the time they were created and once a package maintainer launches a new release, images will automatically download the new release in the next build. *VPR* measures the version pining ratio among all installed dependencies.

$$VPR = \frac{PP}{P} \tag{6.3}$$

PP: the number of "pinned" packages.

P: packages count.

11. **Average Command Count Per Run Instruction (*ACCR*):** *ACCR* measures the average of commands used within a single RUN instruction in a Dockerfile.

$$ACCR = \frac{1}{R}\sum_{i=1}^{R} CR_i \tag{6.4}$$

R: the total number of RUN instructions.

CR: the number of commands per run instructions.

12. **Average Line Count Per Run Instruction (*ALCR*):** *ALCR* measures the average of lines used to write a single RUN instruction in a Dockerfile.

$$ALCR = \frac{1}{R}\sum_{i=1}^{R} LR_i \tag{6.5}$$

R: the total number of RUN instructions.

LR: the number of lines per run instructions.

Table 6.2: Anti-patterns with corresponding metrics

Anti-Pattern	**Metrics**
Big Image	Package Size (PS) Base Image Size (BIS) Final Layers Count (FL)
Hazy Image	Version Pinning Ratio (VPR) Packages Count (P) Instructions Count (I) Base Image TAG Usage (BITU)
Hardcoded Image	Argument and Environmental Variables Usage (AEU)
Blob Image	Process Management Tools Count (PMT) Stage Tree Leaf Count (STL) Stage Count (S)
Messy Image	Stage Count (S) Alias Count (A) Average Command Count Per Run Instruction (ACCR) Average Line Count Per Run Instruction (ALCR)

6.1.3 Implementation of Detection Tool

Following the specification of anti-patterns and Dockerfile-related metrics presented in the previous subsections. This subsection is devoted to describing how the given metrics and Anti-patterns rules were implemented and thus how the detection tool was created.

6.1.3.1 Process **Input:** Rule cards of anti-patterns.

Output: Detection tool for the anti-patterns.

Description: From the Dockerfile, we build an Abstract Syntax Tree (AST), and we parse it to calculate the metrics to model rule cards and detected the five defined anti-patterns in any given Docker project.

6.1.3.2 Implementation In this subsection we use the E-AST described in subsection 5.1.1 in order to collect metrics from Dockerfile, which is a common practice due to the strength of AST structure where comments and layout are meaningless [96]. For this reason, we linked the definitions of a list of metrics previously defined, namely I, PMT, FL, PC, A, $BITU$, AEU, VPR, ACC, R, and $ALCR$ to their corresponding elements in the abstract syntax tree, allowing the metrics to be calculated automatically. Moreover, within a single Dockerfile, we generated a tree to illustrate the hierarchy of the many intermediary images (stages), and we used this specific tree to calculate both S and STL metrics. For the BIS metric, we performed an HTTP POST request to DockerHub API[97] to extract the base image size in megabytes. On the other hand, for PS metric calculation, as there was no API available for package size extraction, we applied web scraping over PKG.ORG[2] website.

Metrics Quantitative Assessment: Dockerfile anti-patterns described in subsubsection 6.1.2.2 are defined using ordinal values of metrics. These values were graded only using a three-tiered scale: HIGH, MEDIUM, and LOW. To tie ordinal values to real metric values while avoiding arbitrary thresholds, we employed the box-plot technique on a dataset of 4,066 Dockerfiles retrieved from BigQuery. For each Dockerfile, we constructed the Abstract Syntax Tree (AST), established the staging tree, utilized DockerHub API calls, and conducted web scraping to compute these metrics.The box-plot, as illustrated in the example of the BS metric in Figure 6.7, divides the data distribution into quartiles, or four equal-sized groupings. The locations of the top (Q1) and lower quartiles (Q3) are shown by

[2]https://pkgs.org/

a box; the inner quartile range (Q2), which is the region between the top and lower quartiles and accounts for 50% of the distribution, is shown by the inside of this box. Lines (also known as whiskers) are drawn all the way to the extremes of the distribution, which are the dataset's lowest and maximum values. Other metrics quartile values are listed in Table 6.3. Using the box-plot methodology, we rated the Dockerfile metrics as high, low, or medium based on the range. LOW values are found below Q1, MEDIUM values are located between Q1 and Q2, and HIGH values are above Q3. These ordinary metrics values were used along with the values of the numerical ones to implement the Dockerfile anti-patterns rule cards.

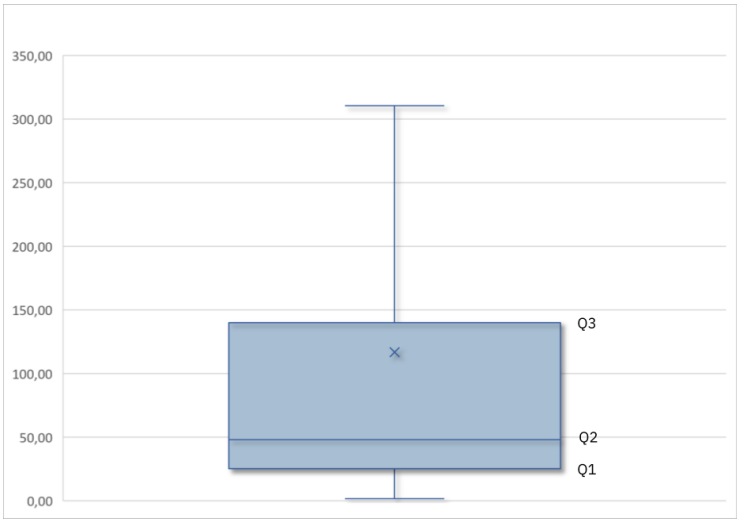


Figure 6.7: *BIS* metric box-plot

6.1.3.3 Tool overview Leveraging the established rules for identifying Dockerfile anti-patterns and the metric thresholds determined by the box-plot methodology, we created an anti-pattern detection tool. The tool offers an intuitive interface that allows users to import a project from a Git repository and then parse it to locate all Dockerfiles. After scanning all the Dockerfiles present within a single project, potential anti-patterns are highlighted and presented in an easy-to-understand format.

Metric	Q1	Q2	Q3
Package Count (P)	2	4	10
Packages Size (PS)	1.3	7.7	38.7
Final Layers Count (FL)	3	5	8
Argument and Environmental Variables Usage (AEU)	0.07	0.22	0.53
Stage Count (S)	1	1	2
Version Pinning Ratio (VPR)	0.06	0.16	0.36
Instructions Count (I)	8	11	16
Base Image Size (BIS)	25.5	48.1	140
Average Command Count Per Run Instruction ($ACCR$)	1.3	2	3.7

Table 6.3: Metrics quartiles

The tool computes a variety of metrics for each Dockerfile in order to assist developers in better understanding its structure and identifying possible problems. These measurements include all the different metrics listed in 6.1.2.3. The tools then uses the rules of anti-patterns along with the calculated metrics to identidy any potenial anti-pattern.

We incorporate project Dockerfiles Table to our tool, as shown in Figure 6.8, to present all the existing Dockerfiles inside a single project. Users can then pick a Dockerfile to display its metrics and anti-patterns with a single click, as illustrated in Figure 6.9. The tool also gives visual representations of each measure, allowing user to see how it compares to best practices and its threshold.

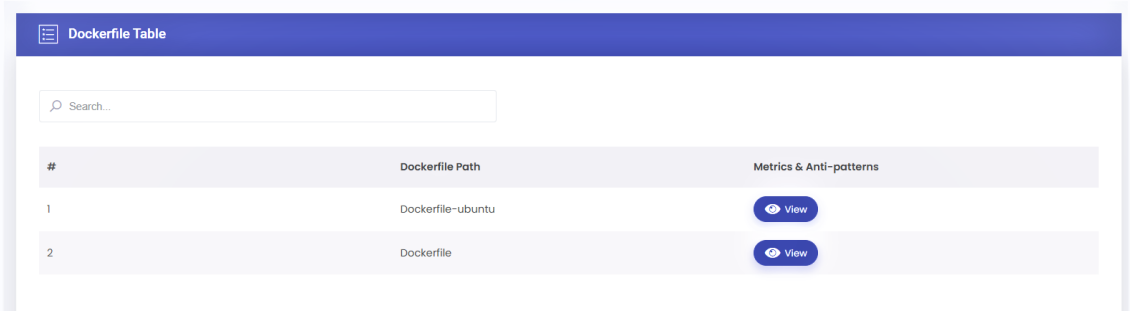


Figure 6.8: Project Dockerfiles table

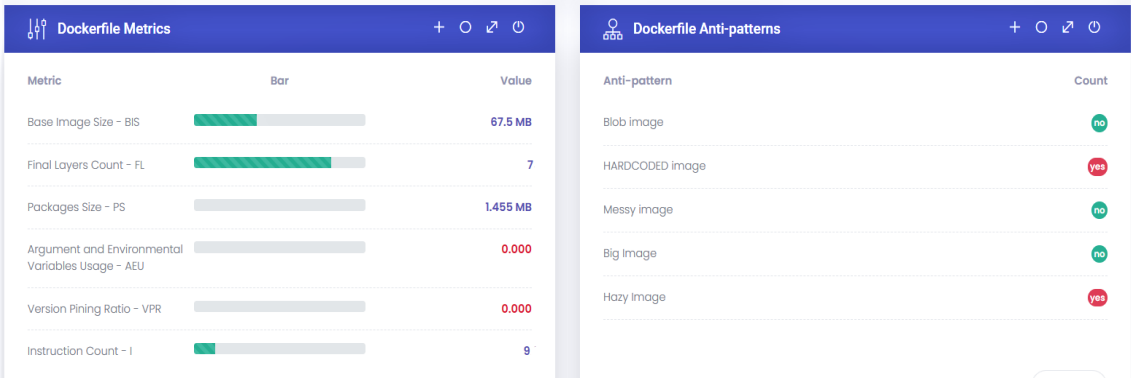


Figure 6.9: Dockerfile metrics & anti-patterns

6.2 Current Results for Anti-pattern Detection Tool Evaluation

6.2.1 Research Questions

RQ1: To what extent do developers assess the accuracy of the defined anti-patterns?

By running a quantitative analysis and investigating the developer's perspective, we can determine the degree to which the observed anti-patterns are correct.

RQ2: What motivates the developer to refactor Dockerfiles, and how important are the detected anti-patterns?

With this question, we look at how important our discovered anti-patterns are in practice by investigating if the anti-patterns have been fixed by developers.

6.2.2 Experimental Design

We consider a total of 92 projects as our validation subjects, containing 104 Dockerfiles. These projects differ in size, age, domain, and other project features. Table 6.4 illustrates the statistics of the demographics of the projects. In this table, we give the average, minimum, and maximum # of contributors, size (in KLOC), # of years from the start of the project to the specified release date, and # of selected Dockerfiles for each project. These projects

comprise a total of 22+ MLOCs with an average evolution history of 6+ years per project. We attempted during the selection process to guarantee the diversity of the study subjects in terms of domain and size to avoid any bias, and we restricted the # of selected Dockerfiles to a maximum of 3 Dockerfiles. We collected the Dockerfiles from the latest releases of the projects. Then, we run our detection tool on the Dockerfiles. The tool delivers the list of detected anti-patterns together with the calculated metrics.

Table 6.4: Statistics of the study subjects

Metric	**Min**	**Average**	**Max**
KLOC	0.199	214.74	4500
HistoryLength (years)	1	6	14
# of collaborators	1	13	30
DFC	1	1	3

Survey Description. To answer RQ1, we asked 27 individuals to assess our approach on 92 selected projects independently. We proceeded with several steps to instruct the participants and avoid any bias. We planned a two-hour presentation about our tool, Dockerfile evaluation in general, and Dockerfile anti-patterns in particular. We also provided them ample time to investigate and try the tool themselves. We created a pre-study survey to assist the participants with running and answering our questions during the pilot sessions. We also acquired their background information and experience with containerization, Docker, etc. We picked the participants based on their years of experience (a minimum of 3 years), current positions (having a current job in the industry), their knowledge of the selected projects, and availability for the manual assessment of our approach. The participants were then divided into groups based on the pre-study survey and their familiarity with the systems under consideration to ensure that all groups had comparable skill levels. The details of

the selected participants may be found in Table 6.5, including their years of programming experience, expertise with Docker, etc.

Table 6.5: Selected participants

System	# of participants	Prog. Exp. (Years) [Min-Avg-Max]	Docker Exp.
Group #1	5	[6.0 - 5.5 - 7.5]	High
Group #2	5	[5.5 - 7.0 - 8.0]	High
Group #3	7	[6.5 - 7.5 - 10.5]	High
Group #4	5	[6.0 - 6.5 - 8.5]	High
Group #5	5	[6.5 - 7.5 - 9.0]	High

During the exercise, we asked each participant to evaluate the correctness and relevance of the anti-patterns detected by our tool on all 92 projects. The participants were provided with a list of defected Dockerfiles and the observed anti-patterns associated with these Dockerfiles. We made sure that each set of anti-patterns had been reviewed by two individuals, and we deemed a set of anti-patterns important if both agreed (the total Cohen's kappa was 0.79). Then, we asked the participants if each anti-pattern was relevant and important. The participants are requested to pick 0 if the discovered anti-pattern is not applicable and incompatible with the Dockerfile, and 1 if the anti-pattern is significant and relevant, and defend their replies for each project. The participants spent 20 minutes, on average, finalizing the survey. Finally, we collected the replies of the participants, and we analyzed the accuracy of our program in detecting Dockerfile anti-patterns.

Evaluation Metrics. We undertake a manual study to evaluate the accuracy of our tool in identifying Dockerfile anti-patterns by comparing the anti-patterns detected by the tool with the ones anticipated by the participants. From the perspective of the participants, for each Dockerfile in our study subject, they evaluate if the anti-pattern is appropriately detected by our tool. To accomplish this, we generate a classification table of the anti-patterns outlined

in our article based on our manual analysis and then examine the findings. The classification table determines from the list of project datasets the anti-patterns that are appropriately discovered by the program as follows:

- An anti-pattern is considered as **true positive** (TP) if it was detected in the Dockerfile by both the tool and the participant.
- An anti-pattern is considered as **true negative** (TN) if it was neither detected by the tool nor by the participant.
- An anti-pattern is considered as **false positive** (FP) if it was not detected by the tool but was identified by the participant.
- An anti-pattern is considered as **false negative** (FN) if it was detected by the tool but was not identified by the participant.

We collected the results after comparing the anti-patterns detected by our tool with those anti-patterns reported by participants (the predicted anti-patterns). Then, we compute the accuracy and recall of the five Dockerfile anti-patterns specified in this study. 310 Dockerfile revisions We report the **precision**[Equation 6.6] as the ratio of the properly detected anti-patterns over the total detected ones and the **recall**[Equation 6.7]as the ratio of the correctly detected ones over the total number of anti-patterns. Then, we calculate the **overall accuracy (ACC)**[Equation 6.8] by dividing the number of successes (true positive and true negative) obtained by the tool by the number of evaluated anti-patterns. We also define a **manual correctness score per project (MCPP)**[Equation 6.9] as the rate of the projects for which the presence and the absence of the 5 anti-patterns were correctly detected by the detection tool.

$$Precision = \frac{\text{detected Anti-pattern} \cap \text{predicted Anti-pattern}}{\text{detected Anti-pattern}} \tag{6.6}$$

$$Recall = \frac{\text{detected Anti-pattern} \cap \text{predicted Anti-pattern}}{\text{predicted Anti-pattern}} \tag{6.7}$$

$$ACC = \frac{\#\text{TP} + \#\text{TN}}{\text{\# of Evaluated anti-patterns}} \tag{6.8}$$

$$MCPP = \frac{\text{\# project with correctly detected anti-patterns}}{\text{\# of Evaluated projects}} \tag{6.9}$$

In general, the anti-patterns fixed by developers in practice are the most critical ones. To assess the importance of the anti-patterns provided in this study (RQ2), we investigated whether programmers sought to correct them using refactorings. Our analysis relies on 166 Dockerfiles provided, each of which contains a series of refactorings made by programmers to correct prevalent Dockerfile-specific issues. This dataset was given by Ksontini et al. [3] and comprises a collection of Dockerfile commits containing Docker-specific refactoring approaches. We investigate the total number of Dockerfiles in the study subject and, for each Dockerfile, we run our tool and we capture the detected anti-patterns in the Dockerfiles before and after applying the refactorings.

6.2.3 Results

Table 6.6: Precision and recall of the Docketfile anti-patterns

Anti-pattern	**# Dockerfiles**	**Precision**	**Recall**
BlobImage (*AEU*)	45	0.971	0.810
HardcodedImage (*S*)	47	0.987	0.872
HazyImage (*VPR*)	74	0.982	0.837
BigImage (*P*)	25	0.694	0.806
MessyImage (*I*)	42	0.94	0.741

RQ1: To what extent do developers assess the accuracy of the defined anti-patterns?

Results for RQ1. Table 6.6 depicts the precision and recall scores for each anti-pattern. Our detection tool was able to detect all five anti-patterns with a precision score of 91.5% and a recall score of 81.4%. On average, 91.5% of the anti-patterns detected by the tool are relevant. Furthermore, 81.4% of the anti-patterns were detected by the tool.

Overall, 94.4% (TP/TP+FP) of the anti-patterns that are actually present in the Dockerfiles were detected correctly by the tool. Furthermore, the tool does not report 76% (TN/TN+FN) of the anti-patterns that were absent in the Dockerfiles. The overall accuracy ACC is estimated to be 89.1% for the detection of the anti-patterns by our tool.

The manual correctness score (MCPP) is 64.4%. Thus, the participants found the detected anti-patterns relatively relevant and consistent with the issues in the Dockerfiles. All participants agreed on the benefits of considering our detection tool. They mentioned that the web interface is easy to use and that the outcomes are interpretable. Furthermore, they highlight that using an automated detection tool that identifies major issues in Docker images significantly increases their trust in managing and deploying the images and would save them time and effort from having to manually inspect the issues.

Finally, we asked the participants to provide their opinions on the usability of the tool. The results show that all participants agreed that the tool is easy to use, and the interface guides the user sufficiently through the tasks. All the information needed for relevant anti-pattern detection was shown on the same screen. Also, it is important to note that the majority of the participants indicated that they did not feel overwhelmed with the amount of data available.

Our answer to **RQ1** is that the five anti-patterns identified in our work are relevant and meaningful. Furthermore, our detection tool significantly reduces the developer's time and effort to discover and fix defects in Dockerfiles.

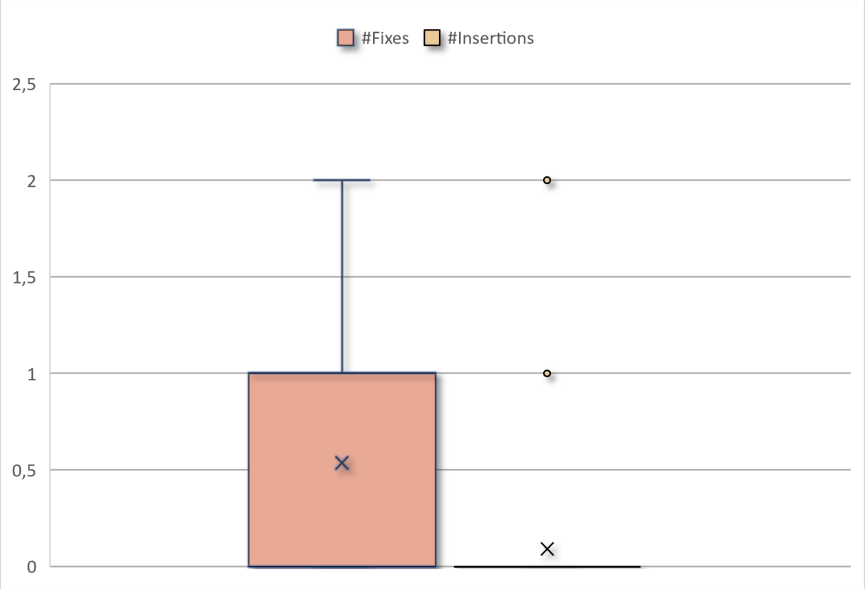


Figure 6.10: Boxplot of #Fixes and #Insertions per Dockerfile

RQ2: What motivates the developer to refactor Dockerfiles, and how important are the detected anti-patterns?

Results for RQ2. we first define two notions. Anti-patterns that have been solved after a set of refactorings are referred to as "fixes". Anti-patterns that have arisen as a result of a series of refactorings are referred to as "insertions". The boxplots of the number of fixes and insertions located in the dataset are shown in Figure 6.10.

Table 6.7 provides samples of the examined Dockerfile refactoring commits. The table depicts the project name, the studied Dockerfiles, the commit ID, and the list of anti-patterns detected by our tool and fixed when applying the set refactorings listed in the last column. The five different types of anti-patterns were fixed by developers at least once. This indicates that, while our defined anti-patterns, which have already been proven correct by RQ1, are relevant, programmers, in general, do not follow the best practices documented in official documents in practice, necessitating studies to identify common flaws and fix issues by suggesting types of refactorings such as those described in Table 6.7.

Table 6.7: Example of anti-patterns detected by the tool and the refactorings applied

Project	**Dockerfile path**	**Commit**	**Fixed Anti-patterns**	**Refactorings**
unbalanced parentheses/ docker-erlang	R10B-6/	9787e1a	Big Image	Update Image Extract Run Instructions
outlierbio/ ob-pipelines	ob_pipelines /apps/fastqc/	5dc54b7	Hardcoded Image Messy Image	Inline Run Instructions Add ARG instruction
collinbarrett/ FilterLists	services/ directory/ src/FilterLists .Directory.Api/	ece877e	Blob Image	Move Stage
CogStack/ cogstack	dockerfiles/ gate/	ae394ce	Hazy Image	Update Base Image TAG

According to the findings, the anti-patterns that have been fixed after applying the refactorings are much greater than the freshly introduced ones. In fact, across the five anti-patterns, we define in this work, the applied refactorings have successfully corrected 0.53 anti-patterns on average per Dockerfile, with a maximum of 2 fixes. When using the refactorings, however, 0.09 anti-patterns were introduced on average per Dockerfile. It's worth noting that a single set of refactorings may fix non-critical flaws in Dockerfiles as well as the most crucial issues that programmers seek to fix. For this reason and in order to complement the findings that answer RQ2, we asked the participants how essential it is to solve each of the co-occurring anti-patterns identified by our methodology. As a consequence, 86% of participants agree that all found anti-patterns that have been corrected as a result of the refactorings are significant and important.

We use a histogram to display the distributions of the anti-patterns that have been repaired (i.e., fixed) across the Dockerfiles in our dataset, as shown in Figure 6.12. We show how many of a given sort of anti-pattern may be found in the Dockerfiles of the study subjects. A large number of fixes for a certain anti-pattern implies a high priority for

repairing it, and hence the high importance of the anti-pattern outlined in our work. Big Images account for 80% (71 out of 89) of the fixes, Hardcoded Images for 15% (13 out of 89), Messy Images were addressed 3 times, and both Blob Images and Hazy Images were solved only once, as shown in Figure 6.12. Big Image anti-patterns and Hardcoded Image are the two most significant anti-patterns of the five anti-patterns. Unsurprisingly, Big Image has received the most fixes. According to our hypothesis, this anti-pattern is the most critical of the five anti-patterns. The reason for this is that Dockerfiles are prone to image size increases due to the layer-based structure of the image. In general, the smaller the image, the quicker it gets uploaded, and the faster it can scale. Developers usually use Docker as a virtualization technique because of its lightweight and scalable properties compared to virtual machines. However, following refactorings, Blob Images and Hazy Images have been addressed only once. The reason for that is that among all the possible refactorings, only two types of refactorings can solve Blob Image or Hazy Image, "Move Stage" for moving or removing a non-needed stage from a specific Dockerfile, and "Update Image TAG" for updating the image tag from the latest to a specific version. Both of these refactorings happened only once in the dataset. Although there is a very low incidence of fixes for these anti-patterns, the detected samples correctly fit our anti-pattern specification. We can take as an example the Blob Image fix provided in listing Listing 8 where we can clearly see that the refactoring applied within this file was aimed at removing the stage of migration testing from the Dockerfile since CI/CD phases should be conducted outside of Docker scope, which perfectly matches our Blob Image anti-pattern definition.

In the same way, we studied the fixes, we present the distribution of the insertions in Figure 6.11. In reality, the overall number of insertions was rather low compared to the number of fixes (15 insertions versus 89 repairs). In particular, Messy Image anti-patterns were the most frequent among insertions, accounting for 60.0% percent (9 out of 15) of insertions. This finding may be explained by the fact that refactoring operations like "Inline

Run Instruction' as a way to reduce the number of layers and thus fix the Big Image anti-pattern might produce Messy image anti-patterns because developers, in most cases, care about regrouping run instructions to reduce the layers but pay less attention to the formatting of the different combined commands. We studied the other minor insertions for Big Image, Blob Image, and Hardcoded Image anti-patterns. We discovered that their existence was mostly connected to specific code modifications conducted inside the same commit but not to refactoring practices.

Our response to **RQ2** is that our tool is capable of detecting Dockerfile issues. All observed anti-patterns were corrected at least once. However, the frequency with which anti-patterns are fixed varies, which can be attributed to the fact that the impact of some anti-patterns, such as the Big Image anti-pattern, is far more critical and costly from the developers' perspective, whereas other anti-patterns are given less importance due to the anti-patterns' non-immediate consequences and perhaps to the lack of knowledge of best practices.

```
diff --git
--- a/services/directory/src/[...]/Dockerfile
+++ b/services/directory/src/[...]/Dockerfile
@@ -26,19 +26,6 @@
 WORKDIR /src/FilterLists.Directory.Api
 COPY directory/src/FilterLists.Directory.Api/. .
 RUN dotnet publish -c Release
                    --no-restore -o /app/publish
-# init Infrastructure.Migrations.Tests
-FROM build AS test-migrations
-ENTRYPOINT ["dotnet", "test", "--logger:trx"]
-
-# restore Infrastructure.Migrations.Tests
-WORKDIR /tests/FilterLists.[...].Migrations.Tests
-COPY directory/tests/FilterLists.[...]csproj .
-RUN dotnet restore
```

Listing 8: Commit `99109b6` from `hexagon`: Refactored Docker-compose file to make "benchmark services" optional

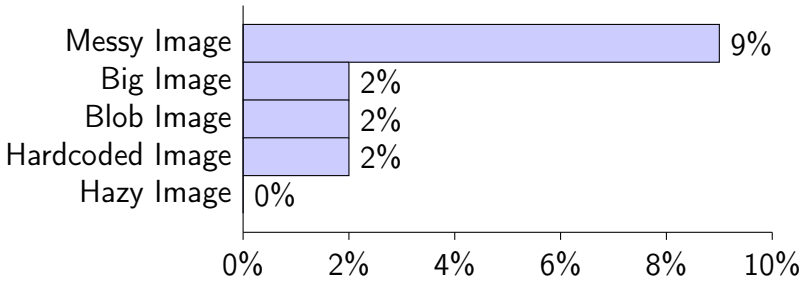


Figure 6.11: Insertions count

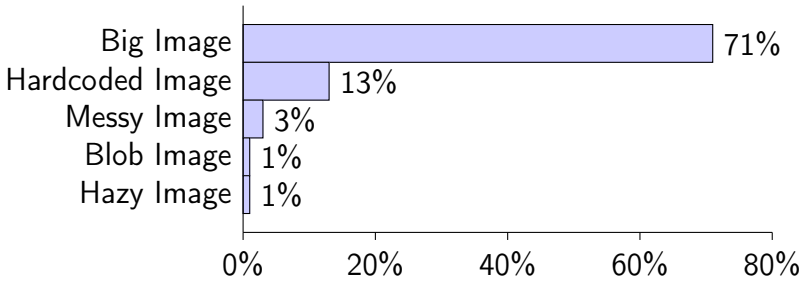


Figure 6.12: Fixes count

6.3 Threats to Validity

The main threat to the validity of our results concerns their external validity, i.e., the possibility to generalize our approach to other Dockerfiles. To address this threat, we ensured that during the evaluation phase, we selected projects that are diverse in domains and sizes and realistic in terms of evolutionary history and popularity. Thus, we used various GitHub metrics,, including the number of contributors, users, stars, commits, etc., to evaluate their popularity and diversity. In future work, we plan to run these experiments on other Docker projects.

For internal validity, the detection results depend on the Dockerfile but also on the anti-pattern specifications using rule cards. We performed experiments on a representative set of anti-patterns to lessen this threat to internal validity. The subjective nature of specifying and validating anti-patterns is a threat to construct validity. We attempt to mitigate this

threat by defining rule cards based on a Domain study of the literature and online materials such as Docker Documentation and by engaging independent engineers with very high programming expertise and experience in Docker throughout the validation.

6.3 Conclusion

In this chapter, we have introduced a methodology for the specification and detection of Dockerfile anti-patterns, addressing a critical gap in the existing literature on Dockerfile quality. Through a structured approach encompassing domain analysis, formal specification, and the implementation of a detection tool, we have successfully identified and formalized five novel Dockerfile anti-patterns: Big Image, Blob Image, Hardcoded Image, Hazy Image, and Messy Image. These anti-patterns represent systemic design flaws that, if left unchecked, can severely compromise the maintainability, efficiency, and performance of Docker-based systems.

Our results demonstrate the effectiveness of the proposed detection tool, with a high precision rate of 91.5% and a recall rate of 81.4%, underscoring its reliability in identifying relevant anti-patterns across diverse Docker projects. The empirical evaluation involving 92 open-source projects and 104 Dockerfiles further validates the practical utility of our approach, with practitioners acknowledging the tool's contribution to improving Dockerfile quality and easing the maintenance burden.

The study also illuminates the interplay between anti-pattern detection and refactoring, a crucial aspect of software maintenance. Refactoring, when informed by precise anti-pattern detection, becomes a potent strategy for mitigating technical debt and enhancing the overall architecture of Docker images. Our findings reveal that certain anti-patterns, such as Big Image and Hardcoded Image, are more frequently addressed by developers, reflecting their immediate and significant impact on Dockerfile performance. Conversely, less frequently

fixed anti-patterns, like Blob Image and Hazy Image, highlight areas where further education and tool support might be necessary to elevate developers' awareness of these critical issues.

CHAPTER SEVEN

REFACTORING AUTOMATION VIA IN-CONTEXT LEARNING

7.0 Introduction

Refactoring, although essential, is often a manual and complex process. A primary concern is managing **image size**—each Dockerfile instruction adds a layer to the final image, and if not optimized, this can result in bloated containers. Effective use of multi-stage builds and careful selection of base images are crucial, but these require significant expertise and effort [11, 7]. Another significant challenge is **maintenance**. Bugs and unexpected behaviors frequently arise, especially when base image tags are not properly specified. Developers who manually add dependencies, often unaware of official trusted images, can introduce inconsistencies that complicate future updates [12]. **Build duration** is another area where technical debt manifests. The order of instructions, complexity of multi-step commands, and the overall number of instructions can significantly affect build times. Poor optimization here leads to longer builds, increased costs, and workflow disruptions [13]. Finally, as projects evolve, Dockerfiles undergo numerous revisions, which can degrade their **understandability** and complicate subsequent updates [14]. The iterative nature of Dockerfile development, coupled with technical debt, makes refactoring essential but increasingly challenging.

Current refactoring tools designed for languages like Java, JavaScript, C, and Python [16, 17, 18, 19] are inadequate for Dockerfiles due to the distinct nature of Infrastructure as Code and their declarative syntax. Dockerfiles are used to containerize applications across various programming languages (e.g., Java, PHP, Go) and domains (e.g., machine learning (ML), web development). Refactoring Dockerfiles involves more than simple structural changes and requires expertise in selecting appropriate base images, tags, and adjusting

dependencies. Techniques such as Extract Stage refactoring add further complexity by necessitating decisions on the optimal number of stages and the selection of files to transfer between them. Consequently, automating Dockerfile refactoring with traditional ML or search-based methods is impractical.

Over the past few years, in-context learning—an approach where LLMs leverage examples and context within input prompts to perform tasks—has proven to be valuable in software engineering tasks like code completion[98], code summarization [99], commit message generation [100], code transformation [101][102], and bug report reproduction [103]. Given their training on vast code repositories, including billions of Dockerfiles created by real developers, we theorize that LLMs can effectively mimic human behavior in writing Dockerfiles and suggest accurate, high-quality refactorings.

In this chapter, we introduce a novel approach that harnesses the capabilities of LLMs and in-context learning 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 LLMs' effectiveness in automating refactoring, using diverse prompting techniques and 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 versus manual refactoring performed by developers, and (4) a thorough examination of the causes of build failures caused by automated and manual refactorings.

7.1 Research Questions

In this chapter, we investigate the practicality and efficiency of automating Dockerfile refactoring and to compare the outcomes with those achieved manually by human developers.

We leverage LLMs and employ in-context learning techniques. To this end, we propose to address the following research questions, which are visually summarized in Figure 7.1.

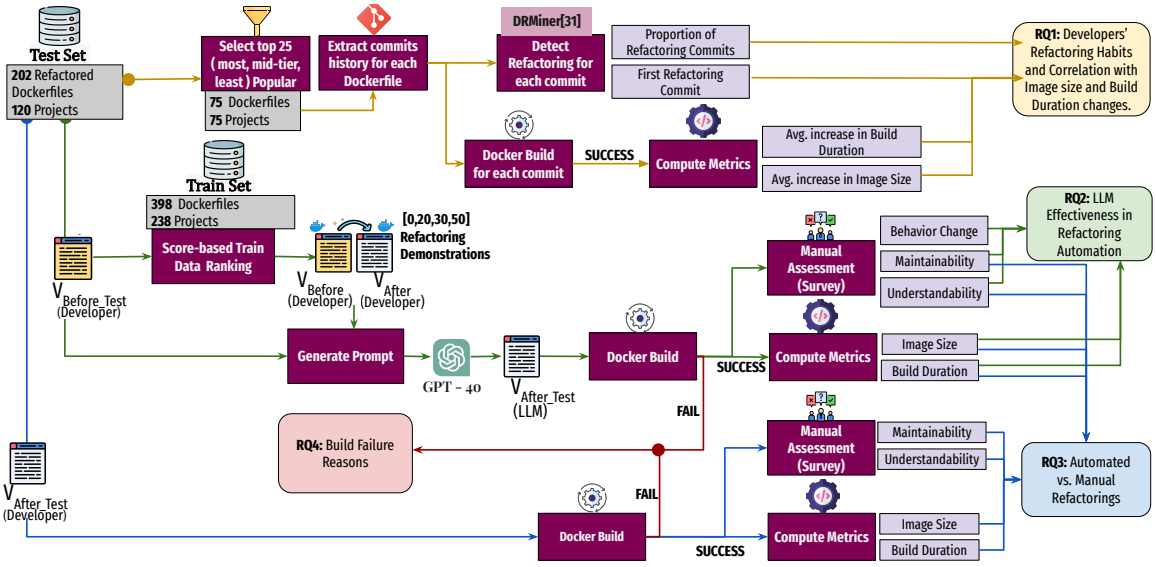


Figure 7.1: Approach overview and addressed RQs (RQ1: yellow arrows, RQ2: green arrows, RQ3: blue arrows, and RQ4: red arrows)

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 7.1 (yellow arrows). For the commit

history of each Dockerfile, we identify instances of refactoring using DRMiner [104]. 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 are LLMs in automating Dockerfile refactoring using zero-shot, one-shot, and few-shot learning approaches?

The objective of this research question is to evaluate the effectiveness and reliability of LLMs in automating Dockerfile refactoring. We are particularly interested in how different in-context learning approaches—zero-shot, one-shot, and few-shot learning—impact the performance of these models.

To answer **RQ2**, we utilize a dataset comprising 202 refactored Dockerfiles from 120 unique open-source projects. As illustrated in Figure 7.1 (green arrows), we start by prompting the LLM with the pre-refactoring version of each Dockerfile ($V_{\text{Before\_Test}}$), 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 draw on recent studies [105, 106, 107] that highlight the importance of demonstration quality for in-context learning. 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 functionality remains unchanged. Finally, we evaluate effectiveness using several key aspects including build failure rates, maintainability, understandability, image size, and build duration.

RQ3: How does the effectiveness of Dockerfile refactorings recommended by LLMs compare to those performed by human developers?

In this RQ, we aim to determine how well Dockerfile refactorings recommended by LLMs compare to those performed by developers. Specifically, we seek to evaluate whether LLMs can match, surpass, or fall short of human performance in these tasks. This evaluation will offer insights into the practicality of automated refactoring in real-world software development.

To answer **RQ3**, we use the results from the best-performing in-context learning approach identified in RQ2. We conduct a detailed comparison of LLM-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 7.1 (blue arrows). This comparison is based on the same effectiveness aspects used in RQ2: build failure, image size, build duration, understandability, and maintainability.

RQ4:What causes build failures in Dockerfile refactorings generated by LLMs, and how do these reasons compare to build failures encountered by human developers?

In **RQ2** and **RQ3**, it was observed that while the LLM achieved commendable improvements in image size, build time, 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 for both LLMs and human developers. 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 LLMs and human developers over a set of 182 Dockerfile refactoring cases. The steps for this analysis are illustrated in Figure 7.1 (red arrows).

7.2 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 are detailed in Figure 7.2.

To begin with, we use the Google BigQuery GitHub Public Dataset [108], 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. [109]. 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

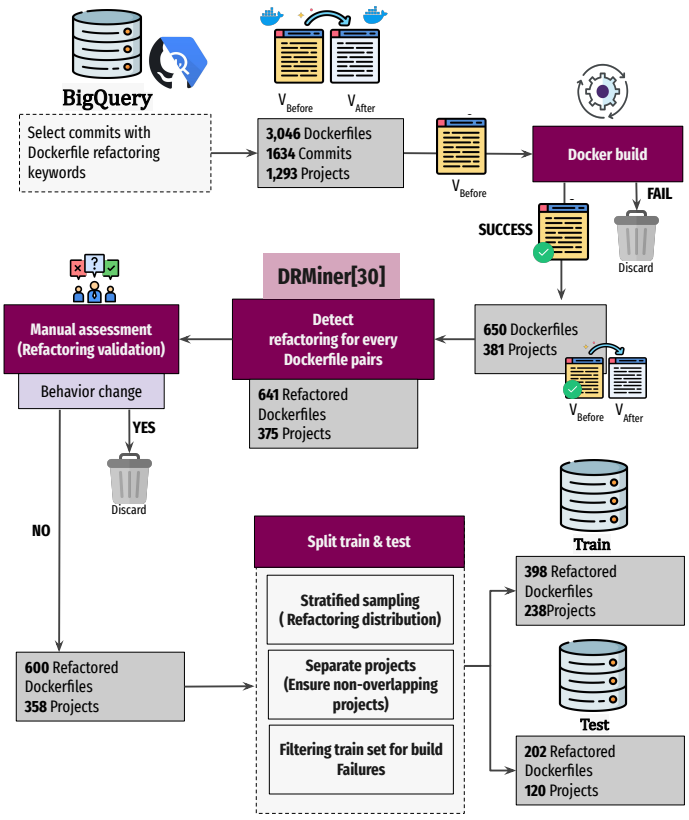


Figure 7.2: Dataset filtering

381 projects. This ensures that we are working with functional Dockerfiles before applying automation.

We process the dataset further using DRMiner [104], 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 three 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 the dataset only reflects refactoring changes, three lab members reviewed the Dockerfile pairs to exclude files with other modifications, such as adding or removing

dependencies or altering the file system, which could affect Docker image size or build duration. After this behavioral assessment, 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**.

7.3 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 possible refactoring actions. $RD = \{V_{\text{Before}_i}, V_{\text{After}_i}, R_i\}_{i=1}^{n}$ 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 size$(P) \leq$ context_window, which means that the prompt should fit within the token limit of the language model.

Figure 7.3 illustrates the prompt template for Dockerfile refactoring automation. In the initial section of the prompt, the model is informed about the specific subtasks it needs to

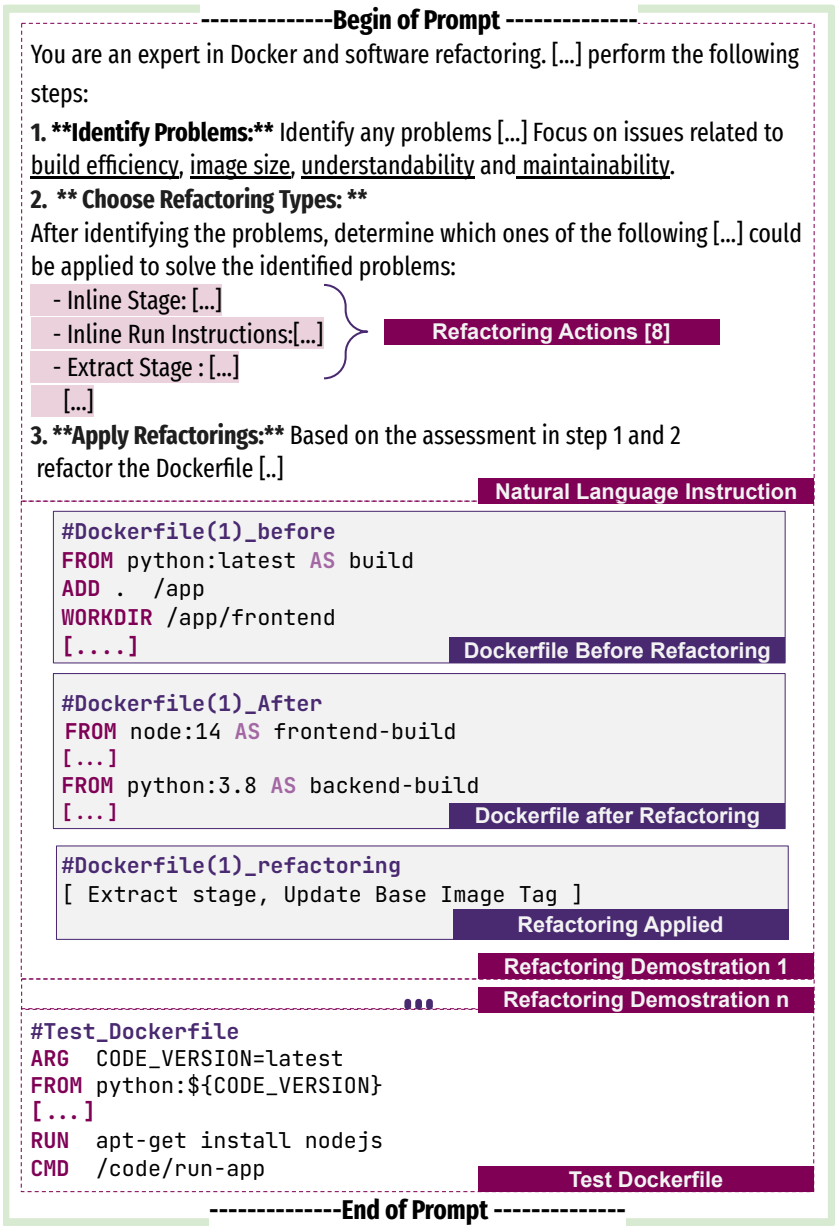


Figure 7.3: Refactoring automation prompt template

follow. Specifically, we first instruct the LLM to identify issues in the Dockerfile related to build efficiency, image size, understandability, and maintainability. Based on the identified problems (if any), we then instruct the LLM to select and apply the appropriate refactorings according to the definitions provided by [3]. Following this, the prompt includes several

refactoring demonstrations to help the language model understand the expected behavior. Finally, the model is instructed to output the desired refactored Dockerfile given the test Dockerfile, as shown at the bottom of the figure.

7.4 Refactoring Demonstration Retrieval

In RQ2, we develop a score-based demonstration selection strategy based on technical debts defined in [3]. For scoring the train dataset, we focus on image size, build duration, maintainability, and understandability.

As illustrated in Figure 7.1. 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 [110].
- **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 time (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 automatically compute the image size, in megabytes (MB), and build duration, in seconds (s), using an automated script. For maintainability and understandability, we employ a manual assessment. The assessment is performed by three lab members.

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. [111].

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, which can differ in terms of applications and commands. Moreover, prioritizing quality improvements enables the model to learn optimal practices from the best examples.

7.5 Evaluation Metrics

To assess the effectiveness of manual and automated refactorings in RQ2 and RQ3, we also rely on the technical debt outlined in [3]. This involves measuring the image size and build time for each Dockerfile version: original, manually refactored, and automatically refactored. For maintainability and understandability, we use the scoring system detailed in section 7.4.

Six industry developers, each with around two years of Dockerfile experience, evaluated the maintainability and understandability. These developers participated through 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.

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 [112].

7.6 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 [113].

Among these LLMs, GPT-4 (Generative Pre-trained Transformer 4) has shown remarkable performance not only in conventional code-related tasks [114] but also in IaC scenarios [115]. Consequently, we have selected gpt-4o-2024-05-13 (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 [111, 116, 106], 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.

7.7 Results

7.7.1 RQ1: Developers' refactoring habits and their impact on technical debts

To answer RQ1, we analyze the evolution of 75 Dockerfiles from 75 open-source projects. These Dockerfiles are selected from our test dataset based on GitHub star counts to ensure representative samples. We include 25 Dockerfiles from most, mid-tier, and least popular projects, covering different languages and domains. The demographics of these Dockerfiles can be found in [112].

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

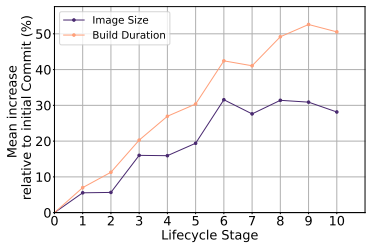


Figure 7.4: Mean Image Size & Build Duration increase relative to initial commit over project lifecycles

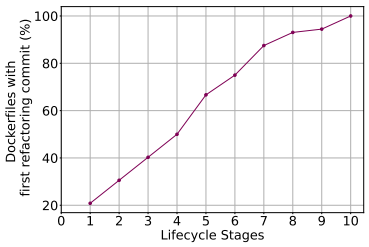


Figure 7.5: Cumulative percentage of Dockerfiles in our dataset with first refactoring commit

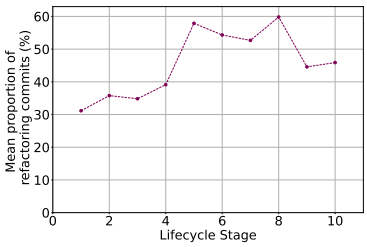


Figure 7.6: Mean proportion of refactoring commits across project lifecycles

these outliers, *our analysis focuses on 72 Dockerfiles, covering a total of 1,459 Dockerfile commits*. Figure 7.4 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.

> **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 7.5 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 7.6 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 time 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.

7.7.2 RQ2: Effectiveness of LLM in refactoring Dockerfile

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 7.1 and Table 7.2, 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.

Table 7.1: Evaluation of Prompting Techniques and Developer Performance (Image Size and Build Duration)

		Image Size				Build Duration			
Prompting Techniques / Developer	Build Success Rate (%)	Improvement Rate (%)	Deterioration Rate (%)	Average Reduction	Total Reduction (GB)	Improvement Rate (%)	Deterioration Rate (%)	Average Reduction	Total Reduction (minutes)
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
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
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
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
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**
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

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.

Table 7.2: Evaluation of Prompting Techniques and Developer Performance (Understandability and Maintainability)

		Understandability		Maintainability	
Prompting Techniques / Developer	Build Success Rate (%)	Improvement Rate (%)	Deterioration Rate (%)	Improvement Rate (%)	Deterioration Rate (%)
Zero-shot	38% (77/202)	**93% (70/75)**	**0% (0/75)**	**99% (74/75)**	**0% (0/75)**
One-shot	47% (94/202)	87% (79/91)	**0% (0/91)**	97% (88/91)	**0% (0/91)**
20-shot	49% (99/202)	80% (79/99)	**0% (0/99)**	96% (95/99)	**0% (0/99)**
30-shot	54% (108/202)	80% (86/108)	**0% (0/108)**	95% (103/108)	**0% (0/108)**
50-shot	**63% (128/202)**	77% (98/128)	2% (2/128)	91% (117/128)	1% (1/128)
Developer	47% (94/202)	56% (53/94)	7% (7/94)	82% (77/94)	6% (6/94)

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 LLM 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. To this end, we employ Spearman correlation analysis [117]. The results show a moderate negative correlation (-0.60) between image size and build duration, indicating that better image size often comes with longer build times. A strong negative correlation (-0.80) between build duration and both understandability and maintainability suggests that reducing build time can harm code clarity and ease of maintenance. Understandability and maintainability have a perfect positive correlation (1.00), meaning improvements in one lead directly to improvements in the other. Conversely, image size changes do not affect understandability and maintainability (0.00). The correlation matrix can be found in [112].

Finding-4. Improvements in image size often lead to longer build times, 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 7.7. 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

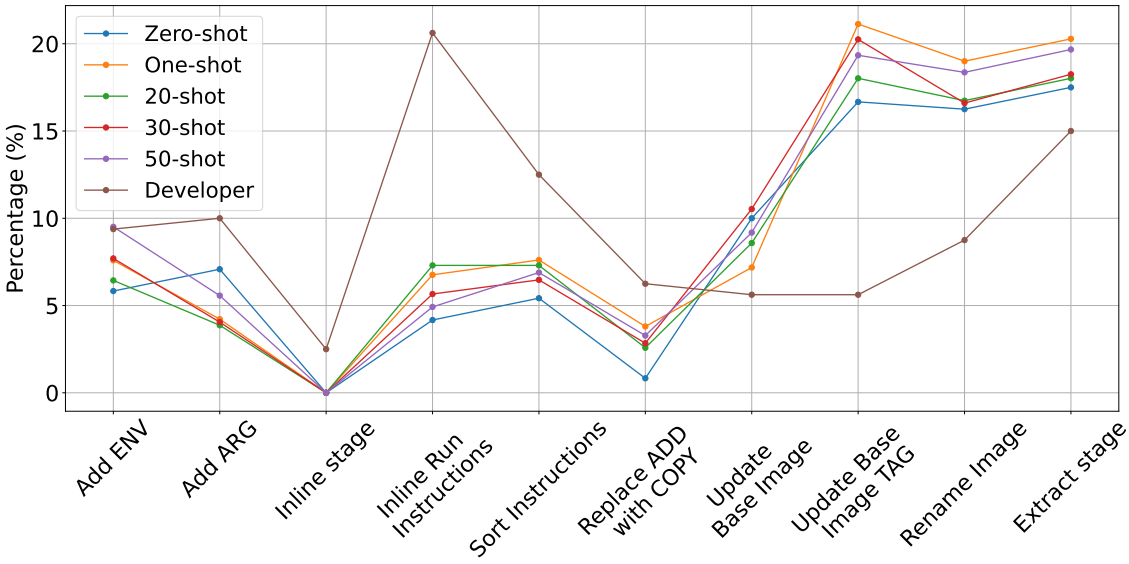


Figure 7.7: Refactoring techniques distribution

(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.** Irrespective 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.

7.7.3 RQ3: LLM-generated vs. human developer refactorings

While metrics show significant improvements through LLM automation in 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 7.1 and Table 7.2. Developers achieved a build success rate of 47% (92/202), while the 50-shot setting achieved 63% (128/202).

For this RQ, to ensure a detailed comparison, we focused on the 66 Dockerfiles that were successfully built by both methods and analyzed improvements in build time, image size, maintainability, and understandability for this intersection set.

The LLM refactoring outperformed developer refactoring, reducing median image size twice as effectively As illustrated in Figure 7.8, 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.

The LLM refactoring achieves a much greater reduction in build duration compared to developer refactoring. As illustrated in Figure 7.8, the original Dockerfiles exhibit a median build duration of 58 seconds. The developer's refactoring maintained the median build time 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.

LLM 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.

LLM refactoring shows a higher maintainability improvement rate compared to developer refactoring, with strong results for both. For maintainability, LLM refactoring

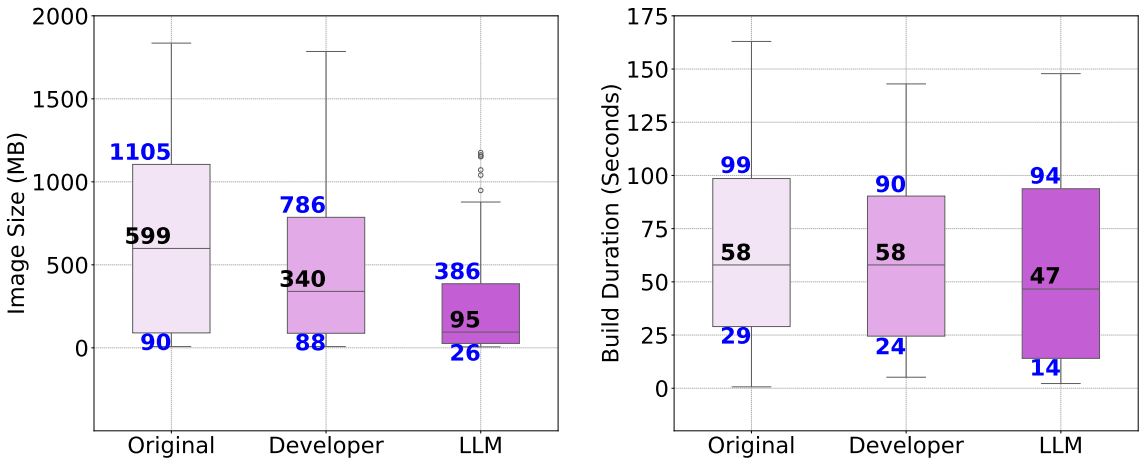


Figure 7.8: Docker image sizes and build durations across three categories: the pre-refactored Dockerfiles (original), those refactored by developers, and those refactored using the 50-shot LLM approach

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.** LLM 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 7.9 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 7.9, we present a case to demonstrate the difference in Dockerfile refactoring strategies between the developer and the LLM. The original Dockerfile (pre-refactoring), using a node:9.11 base image, resulted in an image size of 1110 MB and a build time 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.

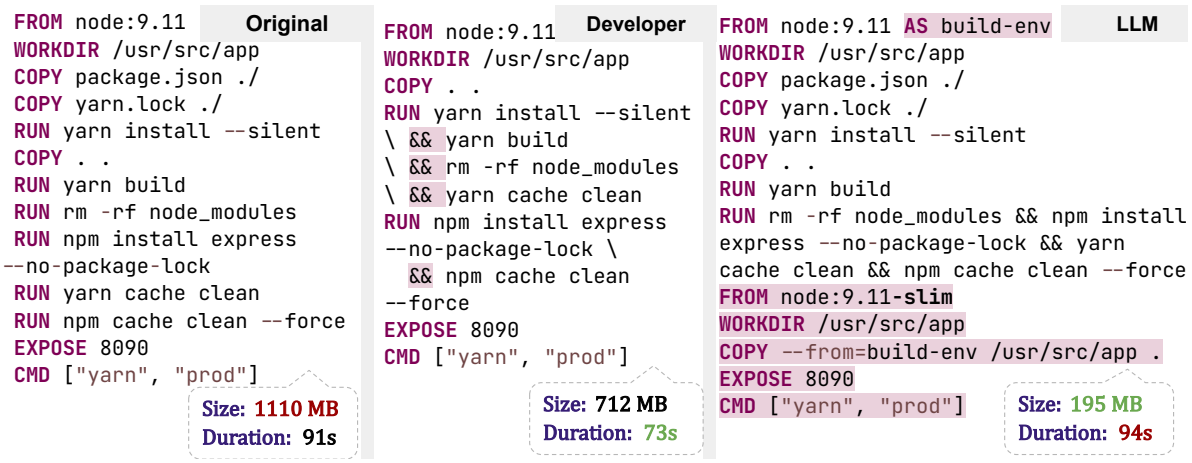


Figure 7.9: Example of manual vs. automated refactoring (Commit 729ee76 from cars10/elasticvue; comments have been removed)

7.7.4 RQ4: Build Failure Reasons

Despite notable improvements in various metrics, build failure rates remain relatively high for both developers and LLMs, 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 [118] , 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 [119], 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, [120], 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 [121] 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.

7.8 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 [104], a state-of-the-art tool, to detect refactorings between commits. While DRMiner has a reported F1 score of 0.94, its limitations may lead to missed or incorrectly identified refactorings.

7.8 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.

CHAPTER EIGHT

CONCLUSION

The features and improvements that were delivered in this dissertation and the results that were achieved are summarized in this chapter. In addition, we suggest possible enhancements to the proposed contributions and discuss them.

8.1 Summary

This thesis focuses mainly on enhancing Docker-based project quality by proposing different techniques and introducing novel tools for refactoring, anti-pattern detection, and the mitigation of technical debt.

In Chapter I, II and III we define the research context and challenges, the contributions of this dissertation, required background, and state-of-the-art and related works to our approaches.

In Chapter IV, we conduct an in-depth empirical study to explore Docker refactorings and technical debts—areas that remain under-explored in current research. By carefully analyzing Docker projects, we define specific refactoring types and identify technical debt categories unique to these environments. Our study also investigates the co-evolution of Docker-specific refactorings alongside traditional software refactorings within the application code that hosts these containers. The outcome of this research is a comprehensive taxonomy comprising 23 Dockerfile-specific refactorings, 18 Docker-compose refactorings, and 9 categories of technical debt. These findings provide crucial insights into how Docker refactorings can be strategically applied to enhance build time, maintainability, and image size for Dockerfiles, while Docker-compose files prioritize improvements in reusability, understandability, security, and extensibility.

The implications of this empirical study will help to (i) understand how and why quality issues and technical debts appear in Docker projects and how refactorings would address those issues, (ii) design new automated tools to integrate novel Docker-specific refactorings, (iii) provide guidelines for best practices, and anti-patterns/bad smells for practitioners in evolving effectively Docker projects, and (iv) assist educators in teaching quality issues related to Docker projects.

In Chapter V, we present DRMiner, a tool designed for Dockerfile refactoring detection and analysis. Built upon an E-AST based component matching algorithm and robust detection rules, DRMiner demonstrates exceptional performance, achieving high precision and recall rates. Its ability to generalize across diverse Dockerfile contexts underscores its relevance in the dynamic Docker landscape.

The tool's ability to process commits offers significant opportunities for empirical researchers to develop precise Docker-specific refactoring datasets, enhances the accuracy of bug-inducing analysis techniques with commit-level refactoring data, and enables automatic documentation of refactoring operations at commit-time for more detailed commit messages. These enhancements will further aid in code review and the comprehension of software evolution.

In Chapter VI, we propose a methodology for the specification and detection of Dockerfile anti-patterns. This chapter outlines a structured approach that combines domain analysis, formal specification, and the development of a detection tool. Through this process, we identify and formalize five novel Dockerfile anti-patterns—Big Image, Blob Image, Hardcoded Image, Hazy Image, and Messy Image. These anti-patterns represent systemic design flaws that can severely impact the maintainability, efficiency, and performance of Docker-based systems if left unchecked. The empirical validation of our detection tool demonstrated its effectiveness in accurately identifying these anti-patterns across various Docker projects.The study also highlighted the critical role of refactoring in mitigating

these anti-patterns. Specifically, anti-patterns like Big Image and Hardcoded Image were frequently addressed by developers due to their significant impact on Dockerfile performance. In contrast, less frequently refactored anti-patterns such as Blob Image and Hazy Image indicate areas where increased awareness and enhanced tool support are needed.

The insights gained from this research not only advance the current understanding of Dockerfile anti-patterns but also provide a practical framework for their detection and resolution.

In Chapter VII, , we explore the automation of Dockerfile refactoring through the use of large language models (LLMs). Analyzing 600 Dockerfiles from 358 open-source projects, we observe that Dockerfile image size and build duration tend to increase as projects mature, often due to delayed refactoring efforts. This chapter presents our novel approach to automating refactoring, leveraging LLMs with a score-based demonstration selection strategy. Our findings indicate that automated refactoring significantly improves Dockerfile quality, particularly in reducing image size and build duration, while also enhancing understandability and maintainability. A comparative analysis revealed that LLM-driven methods outperform manual refactoring, particularly in optimizing image size .

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.

8.2 Future Work

Building on the substantial contributions presented in this dissertation, several avenues for future research and development emerge, each with the potential to further advance the field of Dockerfile quality assessment and refactoring automation.

One promising direction is the enhancement of our anti-pattern, refactoring, and technical debt definitions to encompass a broader range of IaC configurations beyond Dockerfiles and Docker-compose files. Expanding the tool's capabilities to include Kubernetes, Terraform, and Ansible. Additionally, integrating machine learning models to predict the emergence of anti-patterns based on historical data could enable proactive rather than reactive refactoring, thereby preventing the accumulation of technical debt.

Another promising direction worth exploring is the systematic analysis and mitigation of build failures in Docker projects. While our study identified that both manual and automated refactorings can inadvertently introduce build failures, a focused investigation into the root causes of these failures could lead to the development of more resilient refactoring techniques. By understanding and preemptively addressing the dependency issues and context errors that often cause builds to break, we can enhance the reliability of both manual and automated refactoring processes.

Further refinement of automated refactoring methods using LLMs also holds great potential. While our study demonstrates the efficacy of LLMs in optimizing Dockerfiles, future research could explore the implementation of adaptive learning strategies. Such strategies would enable LLMs to continuously refine their suggestions based on real-time feedback from developers, evolving into a dynamic, interactive refactoring tool that grows more sophisticated with each iteration.

Additionally, exploring the trade-offs inherent in Dockerfile refactoring presents a rich area for further investigation. Our findings suggest that improvements in one metric, such as image size, can sometimes negatively impact others, like build duration or maintainability. Future work could involve developing sophisticated multi-objective optimization algorithms that balance these trade-offs more effectively, helping developers make informed decisions that align with their project's priorities.

Bibliography

[1] 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*, pp. 128–130, 2020.

[2] 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)*, pp. 323–333, 2017.

[3] 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.

[4] Z. Lu, J. Xu, Y. Wu, T. Wang, and T. Huang, "An empirical case study on the temporary file smell in dockerfiles," *IEEE Access*, vol. PP, pp. 1–1, 03 2019.

[5] 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.

[6] I. Miell and A. Sayers, *Docker in practice*. Simon and Schuster, 2019.

[7] 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*, pp. 1–12, 2024.

[8] Q.-C. Bui, M. Laukötter, and R. Scandariato, "Dockercleaner: Automatic repair of security smells in dockerfiles," in *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 160–170, IEEE, 2023.

[9] T. Durieux, "Parfum: Detection and automatic repair of dockerfile smells," *arXiv preprint arXiv:2302.01707*, 2023.

[10] 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)*, pp. 231–241, IEEE, 2024.

[11] 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)*, pp. 94–105, IEEE, 2023.

[12] 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.

[13] 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*, pp. 1–13, 2022.

[14] 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)*, pp. 247–256, 2020.

[15] M. Fowler, *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley Professional, 1 ed., 1999.

[16] D. Silva, J. P. da Silva, G. Santos, R. Terra, and M. T. Valente, "Refdiff 2.0: A multi-language refactoring detection tool," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2786–2802, 2020.

[17] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 483–494, IEEE, 2018.

[18] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automatic detection of refactorings for libraries and frameworks," in *Proceedings of Workshop on Object Oriented Reengineering (WOOR'05)*, 2005.

[19] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE transactions on software engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[20] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu, "Emerging trends, techniques and open issues of containerization: A review," *IEEE Access*, vol. PP, pp. 1–1, 10 2019.

[21] Jonathan Johnson, "Devops blog," 2021. `https://www.bmc.com/blogs/what-is-a-container-containerization-explained/`.

[22] Docker, 2020. `https://docker.com`.

[23] DockerHub, 2020. `https://hub.docker.com`.

[24] A. Efe, U. ASLAN, and A. KARA, "Securing vulnerabilities in docker images," *International Journal of Innovative Engineering Applications*, vol. 4, pp. 31–39, 06 2020.

[25] Y. Jiang and B. Adams, "Co-evolution of infrastructure and source code - an empirical study," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, 2015.

[26] Docker Compose, 2020. `https://github.com/docker/compose`.

[27] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, (New York, NY, USA), p. 47–52, Association for Computing Machinery, 2010.

[28] R. Bashir, "Anti-patterns in open source software development," *International Journal of Computer Applications*, vol. 44, no. 3, pp. 23–30, 2012.

[29] Docker, 2021. `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/`.

[30] 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)*, pp. 189–200, IEEE, 2016.

[31] 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.

[32] 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.

[33] 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)*, pp. 1148–1160, IEEE, 2021.

[34] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[35] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoringchallenges and benefits at microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, 2014.

[36] J. Ratzinger, T. Sigmund, and H. C. Gall, "On the relation of refactorings and software defect prediction," in *Proceedings of the 2008 international working conference on Mining software repositories*, pp. 35–38, ACM, 2008.

[37] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of api-level refactorings during software evolution," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 151–160, ACM, 2011.

[38] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pp. 104–113, IEEE, 2012.

[39] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *Software Engineering, IEEE Transactions on*, vol. 31, pp. 429– 445, 07 2005.

[40] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," 07 2006.

[41] V. Cosentino, J. Canovas Izquierdo, and J. Cabot, "Findings from github: Methods, datasets and limitations," 05 2016.

[42] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, (New York, NY, USA), pp. 483–494, ACM, 2018.

[43] A. Stewart, "An Empirical Study of Refactorings and Technical Debt in Machine Learning Systems," 2021.

[44] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 166–177, 2000.

[45] G. Antoniol, M. Di Penta, and E. Merlo, "An automatic approach to identify class evolution discontinuities," in *7th International Workshop on Principles of Software Evolution*, pp. 31–40, IEEE, 2004.

[46] Z. Xing and E. Stroulia, "Umldiff: an algorithm for object-oriented design differencing," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 54–65, 2005.

[47] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 371–372, 2010.

[48] A. Loh and M. Kim, "Lsdiff: a program differencing tool to identify systematic structural differences," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pp. 263–266, 2010.

[49] K. Stroggylos and D. Spinellis, "Refactoring–does it improve software quality?," in *Fifth International Workshop on Software Quality (WoSQ'07: ICSE Workshops 2007)*, pp. 10–10, 2007.

[50] E. C. Neto, D. A. da Costa, and U. Kulesza, "The impact of refactoring changes on the szz algorithm: An empirical study," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 380–390, 2018.

[51] A. Ketkar, N. Tsantalis, and D. Dig, "Understanding type changes in java," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, (New York, NY, USA), Association for Computing Machinery, 2020.

[52] F. Palomba, A. D. Lucia, G. Bavota, and R. Oliveto, "Anti-pattern detection: Methods, challenges, and open issues," *Adv. Comput.*, vol. 95, pp. 201–238, 2015.

[53] K. A. El-Dahshan, E. Elsayed, and N. E. Ghannam, "Comparative study for detecting mobile application's anti-patterns," *Proceedings of the 8th International Conference on Software and Information Engineering*, 2019.

[54] C. Politowski, F. Khomh, S. Romano, G. Scanniello, F. Petrillo, Y.-G. Guéhéneuc, and A. Maiga, "A large scale empirical study of the impact of spaghetti code and blob anti-patterns on program comprehension," *Inf. Softw. Technol.*, vol. 122, p. 106278, 2020.

[55] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the International Conference on Software Maintenance (ICSM '04)*, pp. 350–359, IEEE, 2004.

[56] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[57] M. Afrin, S. A. Asma, N. Akhter, J. H. Ridoy, S. S. Sauda, and K. A. Taher, "A hybrid approach to investigate anti-pattern from source code," *2022 25th International Conference on Computer and Information Technology (ICCIT)*, pp. 888–892, 2022.

[58] S. Saluja and U. Batra, "Assessing quality by anti-pattern detection in web services," *CompSciRN: Other Web Technology (Topic)*, 2019.

[59] A. M. Eilertsen, "Refactoring operations grounded in manual code changes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, pp. 182–185, 2020.

[60] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, pp. 1–61, 2012.

[61] 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)*, pp. 347–356, IEEE, 2012.

[62] 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.

[63] A. S. Nyamawe, H. Liu, N. Niu, Q. Umer, and Z. Niu, "Feature requests-based recommendation of software refactorings," *Empirical Software Engineering*, vol. 25, pp. 4315–4347, 2020.

[64] P. S. Sagar, E. A. AlOmar, M. W. Mkaouer, A. Ouni, and C. D. Newman, "Comparing commit messages and source code metrics for the prediction refactoring activities," *Algorithms*, vol. 14, p. 289, 2021.

[65] 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.

[66] 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.

[67] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, *et al.*, "Self-refine: Iterative refinement with self-feedback," 2023. arXiv preprint.

[68] 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)*, pp. 151–160, IEEE, 2023.

[69] 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.

[70] F. Hassan, R. Rodriguez, and X. Wang, "Rudsea: recommending updates of dockerfiles via software environment analysis," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 796–801, 2018.

[71] C. Abid, V. Alizadeh, M. Kessentini, T. do Nascimento Ferreira, and D. Dig, "30 years of software refactoring research:a systematic literature review," *IEEE Transactions on Software Engineering*, vol. 1, no. 1, 2020.

[72] B. P. Datasets, 2020. `https://cloud.google.com/bigquery/public-data`.

[73] L. Zhang, D. Tiwari, B. Morin, B. Baudry, and M. Monperrus, "Automatic observability for dockerized java applications," *arXiv*, vol. 248, pp. 1–14, 2019.

[74] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: A refactoring reconstruction tool based on logic query templates," pp. 371–372, 01 2010.

[75] P. S. Kochhar and D. Lo, "Revisiting assert use in github projects," in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering (EASE '17)*, pp. 298–307, 2017.

[76] A. J. Viera, J. M. Garrett, *et al.*, "Understanding interobserver agreement: the kappa statistic," *Fam med*, vol. 37, no. 5, pp. 360–363, 2005.

[77] T. Tanaka, H. Hata, B. Chinthanet, R. G. Kula, and K. Matsumoto, "Meta-maintanance for dockerfiles: Are we there yet?," *arXiv preprint arXiv:2305.03251*, 2023.

[78] E. A. AlOmar, H. AlRubaye, M. W. Mkaouer, A. Ouni, and M. Kessentini, "Refactoring practices in the context of modern code review: An industrial case study at xerox," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 348–357, IEEE, 2021.

[79] M. F. Rozi, T. Ban, S. Ozawa, A. Yamada, T. Takahashi, S. Kim, and D. Inoue, "Detecting malicious javascript using structure-based analysis of graph representation," *IEEE Access*, 2023.

[80] D. Inc., "Docker docs." https://docs.docker.com/. (Accessed on 11/17/2023).

[81] D. D. McCracken and E. D. Reilly, "Backus-naur form (bnf)," in *Encyclopedia of Computer Science*, pp. 129–131, 2003.

[82] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, pp. 707–710, Soviet Union, 1966.

[83] 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.

[84] Ksontini, Emna, "Study appendix," 2020. https://sites.google.com/view/ase21-docker-refactorings.

[85] R. K. Yin, *Case study research: Design and methods*, vol. 5. sage, 2009.

[86] 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.

[87] R. Prieto-Díaz, "Domain analysis: an introduction," *ACM Sigsoft Software Engineering Notes*, vol. 15, pp. 47–54, 1990.

[88] 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)*, pp. 247–256, 2020.

[89] K. Kapelonis, 2019. https://codefresh.io/containers/docker-anti-patterns/.

[90] Docker, 2021. https://www.docker.com/blog/intro-guide-to-dockerfile-best-practices/.

[91] Docker, 2021. https://docs.docker.com/get-started/07_multi_container/.

[92] D. McCracken and E. Reilly, "Backus-naur form (bnf)," *Encyclopedia of Computer Science*, pp. 129–131, 01 2003.

[93] N. Moha, F. Palma, M. Nayrolles, B. J. Conseil, Y.-G. Guéhéneuc, B. Baudry, and J.-M. Jézéquel, "Specification and detection of soa antipatterns," in *Service-Oriented Computing* (C. Liu, H. Ludwig, F. Toumani, and Q. Yu, eds.), pp. 1–16, Springer Berlin Heidelberg, 2012.

[94] F. Arcelli Fontana, V. Ferme, M. Zanoni, and A. Yamashita, "Automatic metric thresholds derivation for code smell detection," in *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*, pp. 44–53, 2015.

[95] fatherlinux, 2021. http://crunchtools.com/comparison-linux-container-images/.

[96] G. Fischer, J. Lusiardi, and J. Gudenberg, "Abstract syntax trees - and their role in model driven software development," pp. 38–38, 09 2007.

[97] DockerHub API, 2021. https://docs.docker.com/docker-hub/api/latest/.

[98] M. Ciniselli, N. Cooper, L. Pascarella, A. Mastropaolo, E. Aghajani, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of transformer models for code completion," *IEEE Transactions on Software Engineering*, vol. 48, no. 12, pp. 4818–4837, 2021.

[99] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[100] A. Eliseeva, Y. Sokolov, E. Bogomolov, Y. Golubev, D. Dig, and T. Bryksin, "From commit message generation to history-aware commit message completion," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 723–735, IEEE, 2023.

[101] M. Dilhara, A. Bellur, T. Bryksin, and D. Dig, "Unprecedented code change automation: The fusion of llms and transformation by example," *arXiv preprint arXiv:2402.07138*, 2024.

[102] D. Ramos, H. Mitchell, I. Lynce, V. Manquinho, R. Martins, and C. Le Goues, "Melt: Mining effective lightweight transformations from pull requests," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1516–1528, IEEE, 2023.

[103] S. Feng and C. Chen, "Prompting is all you need: Automated android bug replay with large language models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pp. 1–13, 2024.

[104] E. Ksontini, A. Abid, R. Khalsi, and M. Kessentini, "Drminer: A tool for identifying and analyzing refactorings in dockerfile," in *Proceedings of the 21st International Conference on Mining Software Repositories*, pp. 584–594, 2024.

[105] 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.

[106] 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)*, pp. 2450–2462, IEEE, 2023.

[107] O. Rubin, J. Herzig, and J. Berant, "Learning to retrieve prompts for in-context learning," *arXiv preprint arXiv:2112.08633*, 2021.

[108] Google, "Bigquery github database." `https://codelabs.developers.google.com/codelabs/bigquery-github`.

[109] E. AlOmar, 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)*, pp. 51–58, IEEE, 2019.

[110] 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.

[111] 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?," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 761–773, IEEE, 2023.

[112] E. Ksontini, "Replication package." `https://sites.google.com/view/icse25dra/home`, 2023.

[113] 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.

[114] R. A. Poldrack, T. Lu, and G. Beguš, "Ai-assisted coding: Experiments with gpt-4," *arXiv preprint arXiv:2304.13187*, 2023.

[115] 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*, pp. 57–61, 2024.

[116] Z. Cheng, T. Xie, P. Shi, C. Li, R. Nadkarni, Y. Hu, C. Xiong, D. Radev, M. Ostendorf, L. Zettlemoyer, *et al.*, "Binding language models in symbolic languages," *arXiv preprint arXiv:2210.02875*, 2022.

[117] C. Spearman, "The proof and measurement of association between two things.," 1961.

[118] Ksontini, Emna, "Build context error example, project: zhoumingithub/my-zipkin, commit:4e74b2, dockerfile: docker/dockerfile." `https://github.com/zhoumingithub/my-zipkin/commit/4e74b2`.

[119] Ksontini, Emna, "Dependency errors example, project: mattolson/docker-base, commit:ee4d42, dockerfile: Dockerfile." `https://github.com/mattolson/docker-base/commit/ee4d42`.

[120] Ksontini, Emna, "Syntax errors example, project: ipcjk/ixgen, commit:9e1f90, dockerfile: docker/dockerfile." `https://github.com/ipcjk/ixgen/commit/9e1f90`.

[121] Ksontini, Emna, "Missing base images example, project: ekumenlabs/terminus, commit:e2305d, dockerfile: docker/gazebo-terminus-intel-dev/dockerfile." `https://github.com/ekumenlabs/terminus/commit/e2305d`.