Giovanni Rosa

Assessing and Improving the Quality of Docker Artifacts

PhD Thesis

University of Molise, Italy

03, 2024

University of Molise

Department of Biosciences and Territory

Doctoral Course in Biology and Applied Sciences
Applied Science Curriculum, Cycle XXXVI
S.S.D. ING-INF/05

PhD Thesis

Assessing and Improving the Quality of Docker Artifacts

PhD Coordinator

Prof. Filippo Santucci De Magistris

Tutor

Prof. Rocco Oliveto

Co-Tutor

Prof. Simone Scalabrino

Candidate

Giovanni Rosa

Academic Year 2020/2021

Thesis External Evaluators:

Prof. Paolo Tonella, Università della Svizzera Italiana, Switzerland

Prof. Coen De Roover, Vrije Universiteit Brussel, Belgium

Final Exam Committee:

Prof. Gennaro Parlato, University of Molise, Italy

Prof. Gabriele Bavota, Università della Svizzera Italia, Switzerland

Prof. Fabio Narducci, University of Salerno, Italy

"Aim for the moon. If you miss, you may hit a star."

– W. Clement Stone

Abstract

Docker is the most diffused containerization technology adopted in the DevOps workflow. Specifically, Docker allows shipping applications in Docker images, along with their dependencies and execution environment. Dockerfile is the core component of Docker, which defines how a Docker image should be composed. The literature shows that quality issues, such as violations of writing best practices (*i.e.,* Dockerfile smells), are diffused among Dockerfiles. Smells can negatively impact the reliability of the resulting images, causing building failures, poor performance, and security issues. At the same time, there are no clear directions on how to measure and improve the quality of Docker images. The literature shows that composing high-quality Dockerfiles and Docker images is not a trivial task: There is a lack of knowledge and tools to assess and improve their quality. This thesis aims to advance the state-of-the-art in the assessment of the quality of Docker artifacts (*i.e.,* Dockerfiles and Docker images). The first aim is to understand how developers perceive and address Dockerfile smells, allowing to provide a better support in terms of tools and guidelines. The second, is to investigate how developers improve the quality of Docker images (*i.e.,* build and size), and also find out what quality aspects are relevant for their adoption, over alternates. Finally, this thesis aims at support developers in the automated definition of Dockerfiles, via deep learning, thereby reducing the effort and the introduced quality issues related to their manual composition. The contributions of this thesis lead to several lessons learned: First, not all the existing Dockerfile smells are considered a problem, and their severity depends on the context in which they appear. Second, developers prioritize performance over code quality (*i.e.,* build time, image size, and security issues). Third, selecting the right base image is a crucial aspect to consider when defining a Dockerfile since it has a significant impact on the quality of the resulting Docker image. The achieved results revealed some new research directions regarding the definition of new tools to improve the quality of Dockerfiles and Docker images, along with ways to improve the current catalogs of Dockerfile smells.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Application Context

The DevOps culture is widespread in modern companies: Development and operation teams work together to provide a quick and automated release cycle of software systems [71]. DevOps strongly relies on technologies for automating the build and the deployment of the systems, including practices of Continuous Integration and Deployment (CI/CD) proven to bring shorter release cycles [50]. A famous case is represented by the company *Flickr*: In 2009, they reported that the collaboration between the development (Dev) and operations (Ops) teams allowed them to make more than 10 releases per day, improving also the overall development process and quality.[1] While Flickr was an early adopter at the time, more and more companies adopted those practices during the years. We report five notable cases in Table 1.1. Nevertheless, in 2020, GitLab conducted a survey composed of more than 3k developers for DevOps practices and trends. They reported that 59% of the companies deploy multiple times a day, once a day, or

[1] https://www.youtube.com/watch?v=LdOe18KhtT4

Table 1.1: Notable cases of past adoptions of DevOps practices and related technologies.

Company	**Description**
Adobe	In 2015, the company was adopting an Agile and DevOps business culture to speed up the software delivery of their applications. Migrating to a cloud-based platform, their teams were able to easily push frequent updates of their products [53].
Walmart	In 2011, the company founded the *WalmartLabs* division for technology innovation and development. The mission of the tech team was to bring DevOps to automate and accelerate the application deployment. They also created several open-source tools for that scope [115].
Target	The company have been promoting DevOps for years. In 2014 they arrived to make 80 deployments each week thanks to the adopted DevOps practices for infrastructure automation and release engineering [117].
PayPal	In 2017 the company started to adopt container-based tools in their development cycle. By the end of that year, their first production application was released via containers, gaining a drop in terms of processing unit usage by 50% for QA and 25% for production environment [116].
Primerica	In 2019, Primerica announced the migration of their monolithic architecture to an IBM hybrid cloud-based service using containers and microservices. That combination allowed, among the other benefits, an easy monitoring and recovery of their applications [12, 10].

once every few days, making a 45% increase from the previous year.[2] Also, thanks to DevOps improvements in automation, such as CI and containerization, developers no longer have to deal with manual processes for testing and deployment. The 82% of them reported that they are releasing code more quickly.

One of the main challenges faced during the adoption of those new practices is operating software in a production environment. Among the others, it is very important to make sure that the software system behaves exactly as in a development environment. Virtualization and, above all, containerization technologies, are increasingly being used to ensure that such a requirement is met.[3] It is worth saying that containers should not be confused with microservices: While microservices are an aspect regarding the design of software, containers are a way to package software for deployment. The best combination is to use containers to embed microservices. Thus, containerization technologies are a fundamental part of the release automation of DevOps, allowing developers to take control over the execution environment of their products. In fact, containers allow developers to reduce the risks of issues arising from possible differences between the development/testing environment and the production environment. A recent report on *cloud workload platforms* states that 85% of the organizations will adopt con-

[2] `https://about.gitlab.com/developer-survey/previous/2020`

[3] `https://portworx.com/blog/2017-container-adoption-survey/`

tainerization in production environments by 2025, starting from less than 30% observed in 2020 [32].

Docker[4] is the most popular containerization platform used in the DevOps workflow, being widely used by professional developers according to the recent StackOverflow survey.[5] More in detail, the survey reports that Docker has become the most talked-about containerization technology, becoming the #1 most-desired and most-used developer tool platform in 2023. Docker allows releasing applications together with their dependencies through containers (*i.e.,* virtual environments) sharing the host operating system kernel. When using Docker for a specific software product, developers are required to write a Dockerfile, composed of a sequence of instructions. When executed, these instructions allow to build a Docker image which can be run in one or more lightweight virtual machines (Docker containers). Docker allows to share the Docker images with other developers and to download images from a public repository, called DockerHub.[6] They are also used as a *base image* in Dockerfiles, *i.e.,* a starting point for newly defined Dockerfiles. Since its introduction in 2013 until 2021, Docker counts 4.7M of Desktop installations and 396B image pulls from DockerHub.[7] Since Dockerfiles have a direct impact on the composition of the final Docker image, they impact the overall quality and performance of the resulting application. Also, Docker images are the *mold* used to create containers. Thus, due to their strict relation, we can refer to all of them as *Docker artifacts*.

1.2 Research Probelm

Defining Dockerfiles is far from trivial: Each application has its own dependencies and requires specific configurations for the execution environment, along with a lack of expert developers in this domain [41]. Specifically, they are written in a Domain Specific Language (DSL), having a specific set of rules and keywords.

[4] `https://www.docker.com/`

[5] `https://survey.stackoverflow.co/2023/#section-most-popular-technologies-other-tools`

[6] `https://hub.docker.com/`

[7] https://www.docker.com/blog/docker-index-shows-surging-momentum-in-developer-community-activity-again/

We can see that DSL of Dockerfiles as a "*vacation language*";[8] That is, when people go on vacation in a foreign country, they often learn a few words and phrases to get by. For example, they might learn how to buy an ice cream or ask for directions. Similarly, when developers have to deal with projects involving Docker and other technologies, they might learn just the basics to get their application running, preferring to spend their effort on the main programming language used for the functional logic of the application.

A common quality issues occurring in source code is represented by *code smells* [6]. Code smells are poor implementation choices, that do not follow design and coding best practices, such as design patterns [31]. Usually, they are a symptom of *technical debts*, impacting negatively on the quality of a software system in terms of maintainability [16, 57, 119]. Previous works in the literature [13, 129] introduced the concept of Dockerfile smells, which are violations of best practices, similarly to code smells [6, 30]. The occurrence of smells in Dockerfiles might increase the risk of build failures, generate oversized images, and security issues [13, 140, 44, 137].

Despite there are static analysis tools that help to verify the adherence to writing best practices in Dockerfiles [1, 44], they may not be sufficient to assess the absence of code smells [70], widely diffused in open-source Dockerfiles [13, 66, 27]. It is still not clear how they are perceived by developers and if they consider them as issues to address. Also, there is still a lack of advanced tools to support developers in improving the quality and reliability of containerized applications, such as tools for automatic refactoring of Dockerfile smells [58, 13, 44, 5], and, in general, approaches to measure the quality of functional and non-functional aspects of a Docker image. Those tools not only would help developers to improve the quality of their Dockerfiles, but also help developers to choose and obtain better Docker images, overall. On the other hand, DockerHub offers several sets of functionally equivalent environments (*i.e.,* which provide the same software packages). For example, if developers need an environment based on *Apache Tomcat*, they can choose as a base image either *tomcat* or *bitnami/tomcat*. Those

[8]The analogy of Dockerfiles as a *vacation language* was created during the "Gregorio's Dagstuhl", a little scientific retirement organized during my visit to the University King Juan Carlos, in Madrid. Prof. Gregorio Robles hosted the meeting in *El Escorial* on November 25, 2022.

alternatives can be affected by several quality problems, which smells alone might not be sufficient to capture. For example, previous studies reported the diffusion of security vulnerabilities in Docker images [109, 137, 67]. This is an aspect currently not captured by smells. Even if developers can rely on *official* Docker images, *i.e.,* maintained by Docker, they might not be the best choice for their application. In fact, they are not always free from quality issues [44, 137].

Therefore, the *goal* of this thesis is to advance the state-of-the-art of development and maintenance of quality-aware Docker artifacts by providing developers and researchers with tools and insights related to what quality features and issues characterize them.

1.3 Research Contribution

The research *goal* of this thesis is decomposed in three more-detailed *research objectives*, reported in this section, guiding at an high-level the research contribuitions provided with this thesis. The first contribution aims to advance the current state-of-the-art knowledge of Dockerfile smells. While the side effects on the quality of Docker images is known, it is still not clear how developers perceive smells and, more importantly, if they are willing to address them. This constitutes the first research objective of this thesis:

> **Research Objective 1 (RO1).** How do developers perceive Dockerfile smells?

We aim to put the first stone in that direction by asking expert developers about their perception of Dockerfile smells, in terms of the relevance and impact on the quality of the final Docker images. Specifically, we (i) conducted a survey to measure the relevance of the common smells, and (ii) we proposed a prioritized catalog of 36 Dockerfile smells. We go more in-depth by conducting a second empirical study, in the large, to understand (i) how developers fix Dockerfile smells, and (ii) if they are willing to fix 12 of those smells following the recommendations of an automated refactoring tool. Our work is the first contribution in this direc-

tion by providing a first step towards the development of automatic refactoring tools in the domain of Dockerfile smells.

The takeaway from the first research objective is that only a portion of the smells is considered relevant by developers, and thus requiring effort to be fixed. Mostly, developers prioritize aspects that are more related to the performance and security of Docker images, such as improvements aimed at decrease the size, improve build speed, and avoid security misconfigurations. Those aspects are strictly related with the quality of the final Docker image. Thus, understanding the improvements performed on these that are also strongly related with smells, *i.e.,* image size and build, help to understand how they impact the quality of the final Docker image. This leads to the second research objective of this thesis:

> **Research Objective 2 (RO2).** What quality aspects developers prioritize in Docker images?

We explore this new path by conducting two different studies. The first is mainly focused on the source Dockerfiles, where we extracted and analyzed which changes developers perform on them with the aim of improving the quality of the resulting Docker images. The changes mainly result in the reduction of build time and storage size, tha are aspects characterizing the performance. This study proposes the first detailed taxonomy of 54 performance-related changes in Dockerfiles. We also conducted a preliminary analysis quantifying the concrete impact on the image size and build time, to measure the magnitude of improvement.

At this point, we confirmed that smells are not the only factor in defining a good-quality Docker image. This led us to search for a better measure to assess the quality of Docker artifacts, that is the focus of the second study providing the missing piece to fully address the second reseach objective. In the second study, we reviewed the scientific literature to extract the features and metrics, used to measure different aspects related to the quality of Docker images, to define a taxonomy of *configuration* and *external* aspects measuring quality. Then, we evaluated the relation between those aspects and the *adoption* and *prominence* of a Docker image over the similar alternatives available. The results provide (i) a taxonomy of 17 quality aspects and 28 metrics for Docker images, and (ii) a

measure of their impact on the adoption of ∼300 Docker images (*i.e.,* to indicate those that are relevant).

The common factor regarding the quality issues in the studies conducted so far is that developers have minimal support in defining Dockerfiles, and they have to rely on their experience and knowledge to choose the best Dockerfile meeting their requirements. However, the lack of support along with the limited knowledge of developers in this domain might lead unintentionally to the introduction of smells, along with other secondary issues negatively affecting the quality of the final Docker images. This constitutes the gap that the third research objective aims to fill:

> **Research Objective 3 (RO3).** How can we support developers to define better Dockerfiles in terms of quality?

We want to evaluate the effectiveness of deep learning techniques for the automated generation of Dockerfiles. The T5 model has been proven to be effective in several code-related tasks [79], but it has never been applied to different coding paradigms and, specifically, DSL such as the language behind Dockerfiles. Thus, we empirically evaluated the effectiveness of the state-of-the-art *T5* model in the generation of entire Dockerfiles, given a set of input requirements. We also compared the results with two simpler baseline approaches based on information retrieval and semantic similarity. We observed that there is still room for improvement in that direction since some of the resulting Dockerfiles still require manual fixes to be executable. Nevertheless, the results are promising, opening new research directions in the field of automated generation of Dockerfiles.

1.4 Thesis Outline

This section summarizes the structure of the thesis to provide the reader with a quick overview of the content of each chapter. Chapter 2 provides some background and an overview of the literature by (i) describing more in detail how Docker works, (ii) the concept of Dockerfile smells, and (iii) the related

work in the literature. Then, following the chapters providing the contributions previously defined.

- Chapter 3 - "*Not all Dockerfile Smells are the Same: An Empirical Evaluation of Writing Practices by Experts*" reports the perception of Dockerfile smells by developers, along with a prioritized catalog of smells. This chapter contributes to RO1, and it has been accepted at *MSR 2024*;
- Chapter 4 - "*An Empirical Study on Fixing Dockerfile Smells*" presents an empirical study to understand how developers fix smells and an approach able to fix the most common Dockerfile smells (empirically evaluated via pull requests). This chapter contributes to RO1. It has been accepted as a Registered Report paper at *ICSME 2022* [101]. The final study has been submitted in *EMSE* and it is currently under review (Minor Revision).
- Chapter 5 - "*Improving the Build Time and Size of Docker Images: Strategies from Developers*" reports an empirical study aimed at understanding how developers improve the performance of Docker images, and presents a taxonomy of performance-related changes. This chapter contributes to RO2, and we plan to submit it as a journal paper soon;
- Chapter 6 - "*Understanding What Quality Aspects Characterize the Adoption of Docker Images*" investigates how the quality features impact the choice of a Docker image over another by developers. This chapter contributes to RO2, and it has been published in *TOSEM* [100];
- Chapter 7 - "*Towards the Automated Generation of Dockerfiles*" applies state-of-the-art deep learning techniques and information retrieval to automatically generate Dockerfiles from a set of input requirements. This chapter contributes to RO3, and it has been accepted at *ICSSP 2023* [97].

Finally, Chapter 8 provides the conclusions of the thesis by summarizing the contributions, the lessons learned, and future research directions.

CHAPTER 2

Background and Related Work

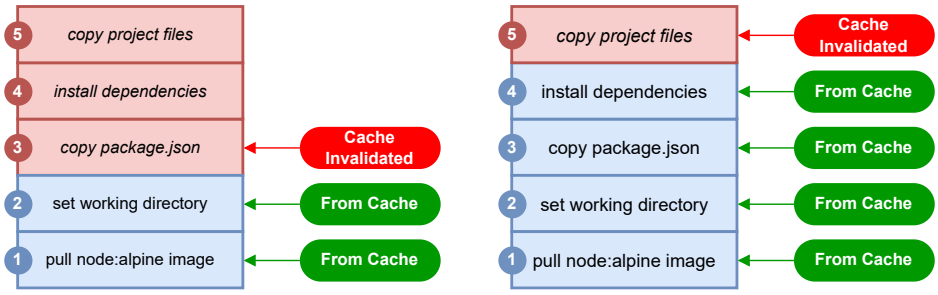
This chapter presents basic concepts about the Docker platform and its features. Then follows an introduction to Dockerfile smells, a set of writing rule violations that can negatively impact the quality of Dockerfiles. Finally, we report the most relevant studies investigating the diffusion of Dockerfile smells. Additionally, we discuss the existing studies investigating the quality of Docker artifacts, and the supporting tools for Dockerfile developers that have been proposed in the literature along with the most commonly used by practitioners.

2.1 Docker Basics

Docker is one of the most popular containerization technologies. Its main purpose is to easily ship an application along with its dependencies and execution environment. The key component of Docker is the Dockerfile: A *blueprint* describing the requirements needed for an application. In fact, it is composed of a set of commands defining the packages and dependencies needed by the application, in addition to the configuration of the execution environment.

```
1 FROM node:17.0.1-alpine
2
3 WORKDIR /app
4
5 # install project dependencies
6 COPY package.json package-lock.json .
7 RUN npm install --production
8
9 COPY myapp/ app
10
11 CMD ["node", "src/index.js"]
```

(a)



(b)

Figure 2.1: Example of a Dockerfile using NodeJS (top), and a representation of how the layer caching works (bottom).

Table 2.1: The Docker instructions that can compose a Dockerfile along with their description.

Command	Description
ADD	Copies files, directories or remote file to the image. Archives are automatically unpacked
ARG	Defines a build-time variable
COPY	Copies files or directories to the image
CMD	Defines the default command and arguments executed when the container starts
ENTRYPOINT	Defines an executable that the container will run with the user provided arguments
ENV	Sets an environment variable <key> to the value <value>
EXPOSE	Acts as a documentation of the network ports on which the container will listen
FROM	Defines the Dockerfile base image
HEALTHCHECK	Tells Docker how to test a container to check that it is still working
LABEL	Adds metadata to an image, such as maintainer and release versions
ONBUILD	Adds a trigger instruction when the image is used as the base for another build
RUN	Executes a command in a new layer on top of the current
SHELL	Overrides the default shell used by the instructions
STOPSIGNAL	Sets the system call signal that will be sent to the container to exit
USER	Sets the system user as which the next commands are executed
VOLUME	Creates a mount point for externally mounted volumes
WORKDIR	Sets the working directory for the successive instructions

The commands are defined using a Domain Specific Language (DSL) composed of a set of Docker-specific instructions [20] and shell script code. Each Docker instruction can perform one or more actions, depending on the command used and the shell code. Fig. 2.1 reports an example of Dockerfile.

The first instruction reported in the example is the FROM commands, *i.e.,* the main Docker instruction with which each Dockerfile begins. It indicated a so-called *base image*, the starting point from which the environment defined in the Dockerfile can inherit dependencies and configurations. The scratch image[1] is a super minimal *base image*, which is the base element of all Docker images. Basically, it is empty: It contains a binary to create a minimal container, and serves as a starting point for building containers. Every Docker image can be used as *base image* and, therefore, extended. The RUN instruction contains one or more commands that will be executed in a shell environment (*i.e.,* RUN <command>), that is by default /bin/sh -c. The complete list of the Docker instructions is reported in Fig. 2.1. Note that the MAINTAINER instruction, which was used to define the Dockerfile author, has been deprecated [21], and thus should be avoided.

[1] https://hub.docker.com/_/scratch/

The step after the definition of a Dockerfile, follows the creation of the Docker image, containing the software application, via the *build* operation. The build process executes all the instructions in the Dockerfile (*e.g.,* to download dependencies and resources or to build the software product). Each instruction produces a *layer*, so a Docker image is composed of stacked *layers*. Each of them is a partial snapshot of the image during the build process. The main purpose of *layers* is to make the build process modular, allowing to speed up it using caching: Instead of running all the instructions of a Dockerfile, it is possible to save time and resources by re-using pre-built layers, when possible (*e.g.,* avoid re-installing of software packages). Fig. 2.1 reports a detail of the layer caching system. The example reports a Dockerfile using `NodeJS`. Thus, when the dependencies of `package.json` file are changed, the image will be rebuilt from that point. Instead, if only the source of the application is changed, only the cache of the final layer is invalidated and then rebuilt. Docker images are executed as *containers*, *i.e.,* a lightweight virtual machine that has its own resources, such as networks and storage volumes.

Each Docker image is uniquely identified by the *digest*, a hash value computed at build time based on the composition of the image. However, it is common practice to assign a meaningful name (*i.e.,* a *tag*) to the images, so that it is possible to refer to them more easily. The image *tag* is usually composed of the name of the software installed in it (*e.g.,* `php`), its version (*e.g.,* 7.0), and its *flavor* (*e.g.,* slim).[2] The latter might denote differences in terms of non-functional requirements (*e.g.,* the reduced size). An example of a tag is `"python:3.11-slim"`, where `python` is the name of the image, while `3.11-slim` is the tag. It is worth specifying that the same Docker image can have multiple tags, thus the only way to identify unique images is using the *digest*. When a tag is not provided, Docker sets by default `latest`. Also, this means that the same image can be built multiple times, each one with a different *digest*.

Similarly to software dependency management systems (such as Maven), all the Docker images are distributed through *registries*, from which developers can retrieve and use them. There are two kinds of registries: private and public. Private registries are usually restricted to specific companies or usages (*e.g.,* an

[2] `https://docs.docker.com/engine/reference/commandline/tag/`

internal registry of a large software system to host and deploy images on Kubernetes), while the main public registry is DockerHub.[3] There are four types of images hosted on DockerHub. First, we have official images,[4] maintained following the official images review guidelines [19]. The aim is to ensure a good quality level of such images, in terms of composition and best practices. Second, there are images from verified publishers, *i.e.,* publishers that can be trusted, but that do not necessarily produce official images that follow the previously mentioned guidelines. Third, we have images that are part of the Docker Open Source program,[5] maintained by organizations that are members of that program. Last, we have the non-official images, which are provided by the users of the Docker community.

The operation of uploading an image on DockerHub is called *push*. It can be performed using the command *docker push*, where usually the developers build the Dockerfile, assign a tag to the resulting image, and upload it to the registry. This means that the Dockerfile is not uploaded to the registry but only to the resulting image blob. In some cases, the developers that maintain the DockerHub repositories add the link to the source Dockerfiles for the image or else the *git* repository where the Dockerfile is maintained. For each hosted image, DockerHub provides a series of information such as tags, last update, *digest*, description, and some metadata such as stargazers count (set by users) and the number of pulls (*i.e.,* how many times the image has been downloaded).

To easily manage multi-container applications, the Docker platform includes the *Docker Compose* tool.[6] In detail, by defining the container configuration using a *docker-compose.yml* file, it allows to easily orchestrate the lifecycle of the containers by starting, stopping, and restarting them. It is very helpful when the software application is composed of multiple services, such as a web application and a database. Moreover, for the orchestration of more complex applications, Docker provides *Docker Swarm*,[7] a container orchestration tool that allows to

[3] https://hub.docker.com/
[4] https://docs.docker.com/docker-hub/official_images/
[5] https://www.docker.com/community/open-source/application/
[6] https://docs.docker.com/compose/features-uses/
[7] https://docs.docker.com/engine/swarm/

manage a cluster of Docker engines. It is a lightweight and simpler alternative to other container orchestration tools, such as *Kubernetes*.[8]

2.2 Dockerfile Smells

A common practice in software engineering is to follow best coding practices aiming at provide the best quality for a software product [74]. This is also true for Dockerfiles, where the Docker community provided a set of writing best practices to follow.

Docker reports an official list of best practices for writing Dockerfiles [18]. Such best practices contain also indications for writing the shell script code included in Dockerfile instructions. For example, they suggest the usage of the dedicated instruction `WORKDIR` instead of the bash command `cd` to change directories. The violation of such practices led to the introduction of the so-called *Dockerfile smells*, inspired by the concept of *code smells* occurring in common source code [30]. Specifically, a Dockerfile smell indicates those instructions of a Dockerfile that violate the writing best practices. Most important, the presence of Dockerfile smells can negatively affect the quality of them [13, 129], since they have a direct impact on the behavior of the deployed software. For example, previous work showed that missing adherence to best practices can lead to security issues [137], negatively impact the image size [44], increase build time and affect the reproducibility of the final image (*i.e.,* build failures) [13, 140, 44].

`Hadolint` is the most popular tool for detecting Dockerfile smells, used by both developers and several studies in the literature [13, 66, 27]. In fact, the occurrence of Dockerfile smells is the main metric to assess the quality of a Dockerfile. The rules enforced by the tool have been defined by the `hadolint` community, extending those reported in the official writing guidelines provided by Docker, with a set of writing rules directly suggested by developers (*e.g.,* by submitting a pull request[9]).

More in detail, the tool detects two kinds of writing violations: (i) errors in Dockerfile instructions, and (ii) *shell* script code-related errors. All the `hadolint`

[8] `https://kubernetes.io/docs/home/`
[9] `https://github.com/hadolint/hadolint/pull/114`

Figure 2.2: Some common Dockerfile smells (left), and their resolution (right).

rules are identified by a prefix followed by a number (*i.e.,* `DLXXXX` or `SCXXXX` for the two previously mentioned categories). There are a total of 66 rules reported in the official documentation [18]. One of the most known rule violations is the missing version tag for the *base image*,[10] which could lead to unpredictable behavior since it impacts the build reliability. Simply, when a new version of the *base image* is released, the Dockerfile will use the latest version, which might not be compatible with the installed dependencies. The most simple example is a Dockerfile using a `nodejs` base image. Newer versions of the *base image* might not be compatible with the `node` packages defined in the `package.json`, and it could lead to build failures. This kind of violation is known as version pinning smell, which consists of the missing version number of dependencies, such as packages and the base image. For example, when using an `ubuntu`-based image, the version number should be specified to pick the specific version wanted by the operating system. This means that the instruction `FROM ubuntu:20.04` should be preferred to `FROM ubuntu`. The same is true when installing dependencies using `apt-get`, where the version number should be specified to ensure the installation of the version of the package. This is because the latest version might not be compatible with the execution environment or the other dependencies, leading to future build reliability issues. Other violations are related to the instruction used for documentation. Examples are the usage of `MAINTAINER` instead of `LABEL` to define the author of the Dockerfile, that has been deprecated. Another group of

[10] `https://github.com/hadolint/hadolint/wiki/DL3006`

writing best practices are those related to image optimization, such as merging consecutive `RUN` instructions into a single one to reduce the number of layers. Other rules are aimed at avoiding space wastage, such as removing `apt` package cache when installing dependencies. Other examples are violations related to the improper usage of instructions. An example is using `COPY` instead of `ADD` to copy files and folders, as the latter is more powerful and can also download remote files and automatically extract archives. However, these implicit actions can lead to unexpected behaviors. Fig. 2.2 reports an example of a Dockerfile affected by smells and its fixed version.

2.2.1 Diffusion of Dockerfile smells

Several studies in the literature investigated the diffusion of Dockerfile smells. A general overview on occurrence of Dockerfile smells was proposed by Cito *et al.* [13] performed an empirical study to characterize the Docker ecosystem in terms of quality issues and the evolution of Dockerfiles. In particular, they analyzed a dataset of 70,000 Dockerfiles extracted from GitHub to characterize the Docker ecosystem in terms of quality issues (smell), extracted using `hadolint`. They found that the most frequent smell regards the lack of version pinning for dependencies (28.6%), which can lead to build failures in the future. In fact, from a sample of 560 projects, they were not able to build 34% of the related Dockerfiles. In the same direction, Wu *et al.* [129]. They performed an empirical study on a large dataset of 6,334 projects to evaluate which Dockerfile smells occurred more frequently, along with coverage, distribution, and a particular focus on the relation with the characteristics of the project repository. They found that nearly 84% of GitHub projects containing Dockerfiles are affected by Dockerfile smells, where the Docker-related smells are more frequent than the shell-script smells.

Lin *et al.* [66] conducted an empirical analysis of Docker images from DockerHub and the git repositories containing their source code. They investigated different characteristics such as base images, popular languages, image tagging practices, and evolutionary trends. The most interesting results are those related to Dockerfile smell prevalence over time, where the version pinning smell is still the most frequent. On the other hand, smells identified as DL3020 (*i.e.,* `COPY/ADD` usage), DL3009 (*i.e.,* clean apt cache), and DL3006 (*i.e.,* image version pinning)

are no longer as prevalent as before. Furthermore, violations DL4006 (*i.e.,* usage of `RUN` pipefail) and DL3003 (*i.e.,* usage of `WORKDIR`) became more prevalent. Therefore, there is a decreasing trend over time, suggesting that developers are more and more aware of the presence of writing rule violations. Eng *et al.* [27] conducted a revisited analysis on the largest dataset of Dockerfiles, spanning from 2013 to 2020, having over 9.4 million unique Dockerfiles. They performed a historical analysis of the evolution of those Dockerfiles, reproducing the results of the previous studies on a larger dataset. Also in this case, the authors found that smells related to version pinning (*i.e.,* DL3006, DL3008, DL3013, and DL3016) are the most prevalent. In terms of Dockerfile smell evolution, they show that the count of code smells is slightly decreasing over time, thus hinting that developers are becoming more aware of them. Still, it is unclear the reason behind their disappearance, *e.g.,* if developers fix them or if they get removed incidentally.

On the other hand, other studies in the literature proposed different Dockerfile smells catalogs from `hadolint`, and also new approaches for detection and fixing. Henkel *et al.* [44] proposed `Binnacle`, a tool that enforces a set of writing rules extracted from the official Dockerfiles written and maintained by developers from Docker. In detail, the approach applies rule-mining to capture the writing patterns found in the processed Dockerfiles, defining in this way a set of *gold writing rules* to which Docker developers adhere in their code. A small part of them is common with the `hadolint` rules. For example, the *aptGetInstallUseNoRec* rule checks for the presence of the flag `--no-install-recommends` to avoid installing additional unwanted packages when using *apt install*. The same check is performed by `hadolint` with rule DL3015. A different example is the *wgetUseHttpsUrl* rule, not supported by `hadolint`, which checks for the usage of an HTTPS URL with *wget* to download external sources. The issue is that using a plain HTTP URL could lead to security issues. An extended set of these rules is proposed by Zhou *et al.* [146]. They introduced *DRIVE*, an approach for rule mining and smell detection in Dockerfiles, supporting a total of 34 semantic and 19 syntactic rule violations. Also, 9 of the proposed semantic rules are not supported by any of the existing approaches.

A different contribution is provided by Lu *et al.* [70], which conducted a case study on a new type of smell, called *temporary file smell* occurring in Dockerfiles.

The smell occurs when temporary files, created during the build process, are added and removed in different layers. This leads to redundancy because files created in one layer are also retained in successive layers until they are deleted. This smell can lead to larger images, negatively impacting the overall efficiency of Docker images. In addition, the authors reported four distinct patterns in which the smell occurs and proposed a state-dependent detection approach to detect the smell, along with three different methods to fix it. A successive work conducted by Xu *et al.* [132] performed an empirical study on the same smell by proposing two approaches to detect temporary file smells using dynamic and static analysis, respectively.

The first study aimed at fixing issues in Dockerfile code was proposed by Henkel *et al.* [45]. In particular, they presented `Shipwright`, a human-in-the-loop system allowing to fix bugs and issues in Dockerfiles. However, the approach mainly focuses on fixing build failures and is not specifically aimed at fixing Dockerfile smells. Chapter 4 reports the first contribution in the scientific literature investigating how developers fix Dockerfile smells. Specifically, the research design has been presented as a registered report paper [101]. Several works followed the same direction, adopting the same validation procedure (*i.e.,* via submitting pull requests), which was in turn inspired by a previous work conducted on a different context, *i.e.,* CI/CD smells [125]. An example is `Parfum`, an approach proposed by Durieux *et al.* [23], which is able to detect and fix Dockerfile smells based on a novel Dockerfile AST parser. `Parfum` in mainly based on the writing rules defined by Henkel *et al.* [44], but also includes some of `hadolint`. Thus, the tool is able to detect and fix a total of 32 Dockerfile smells. In the same direction, Bui *et al.* [9] proposed `DockerCleaner`, an approach to detect and fix 11 security-related bad practices, inspired by a set of known best practices guidelines from the gray literature [84, 106]. Some of them are in overlap with those included in `hadolint`, such as using `COPY` instead of `ADD` or avoiding keeping `root` as the last image user in Dockerfiles.

Even if different approaches and catalogs are available in the context of Dockerfile smells, `hadolint` is currently the most popular and adopted tool in practice by developers; it has 8.8k stars on GitHub at the time of writing. The rules captured by `hadolint` are more oriented towards the composition of the Dockerfile

itself, which is what we are interested in. On the other side, the catalogs proposed in `Binnacle` and `DRIVE` contain mostly semantic rules capturing scripting practices and tool usage. For example, rules like DL3059 (multiple consecutive `RUN` instructions) or DL3020 (prefer `COPY` over `ADD` for files and folders), included in the `hadolint` catalog, are not present in *Binnacle* [43] or *DRIVE* [146]. It is worth saying that `DRIVE` and `Binnacle` are not as well-known by the community. In fact, when looking at the public repositories of those tools[11],[12] they are less popular (*i.e.,* lower number of stars) compared to `hadolint`. Thus, even if their source code is publicly available on GitHub, they are still far away from being widely adopted by developers. Interestingly `hadolint` is also adopted by the maintainer of the official Docker images hosted in DockerHub. If we consider the Docker image of the `kong` API gateway, hosted in DockerHub,[13] and we look at the source Dockerfile hosted on GitHub [14] we can found the comment line `"# hadolint ignore=DL3015"`: This is a special comment line that allows ignoring a specific rule, in this case, DL3015 (*i.e.,* Use `--no-install-recommends` to avoid installing additional packages), which probably is not considered a best practice by the maintainer of the image.

However, the missing point of the existing studies investigating Dockerfile smells, is that no one evaluated them from the point of view of developers, *i.e.,* how they perceive the presence of smells and how and what of them they fix them. The work presented in this thesis investigates this aspect. In particular, in Chapter 3 we investigated the perception of expert developers on Dockerfile smells, specifically by considering the writing pattern that they capture. Additionally, in Chapter 4, we empirically investigate the willingness of developers to fix Dockerfile smells, and how they fix them.

[11] `https://github.com/zwlin98/DRIVE`
[12] `https://github.com/jjhenkel/binnacle-icse2020`
[13] `https://hub.docker.com/_/kong`
[14] `https://github.com/Kong/docker-kong/blob/e4ba2e3/ubuntu/Dockerfile`

2.3 Different Perspectives on the Quality of Docker Artifacts

Even if the occurrence of Dockerfile smells is the most common metric used to measure the quality of Dockerfiles, other aspects should be considered. In fact, several studies in the literature investigated the quality of Docker artifacts from different perspectives, such as the image size, build time and failures, security vulnerabilities, along techniques aimed at their improvement.

2.3.1 Studies on the Performance of Docker images

Improving the quality of Docker artifacts means also improving their performance. Several studies in the literature are focused on the performance of Docker images. Specifically, they consider aspects such as the build time of the source Dockerfile and the final image size (measured as both storage size and number of layers).

In particular, the number of layers can directly impact the final size of the Docker image and the overall build time. In some cases, a large image size could be a symptom of a poorly optimized Docker image, thus revealing bad quality (as we show in Chapter 6).

Previous studies reported that developers pay attention to this aspect: In particular, they are led to change their CI/CD pipelines due to the slow build speed of the embedded Docker images [140]. In this direction, Huang *et al.* [48] proposed `FastBuild`, an approach to speed up the build time of Docker images by maintaining a local file cache to reduce the resources spent in downloading.

As reported in Section 2.2.1, several studies investigated the occurrences over time of smells [66, 27]. Even if smells are widely diffused, there is a declining trend in their occurrence along with the storage size of the resulting images. It has been proved that fixing smells can reduce the space wastage of containers [23]. This aspect is also reported by developers as technical debt in Dockerfiles (Azuma *et al.* [5]). Often, this space wastage includes unnecessary files and dependencies that should be removed. We can conclude that developers pay particular attention to the image size and wasteful resources in Dockerfiles.

The practice of removing unnecessary resources is commonly applied to source code, known as *debloating* [107]. Other approaches have been presented applying the concept of debloating to software containers. Examples are the works of Skourtis *et al.* [111] and Jiang *et al.* [52] that, by acting directly on the organization of the layers in containers, they aim to reduce layer redundancy and space wastage. In addition, Rastogi *et al.* [92, 93] proposed techniques leveraging dynamic analysis to identify only the necessary resources of a given container in order to allow removing the unnecessary ones. We believe that working at Dockerfile-level allows developers to have more control over the resulting Docker images, making them reproducible and more efficient.

However, none ho these studies performed an extensive investigation on what are the changes that can help developers improve the build time and reduce the image size of Docker images. Specifically, improving those aspects by acting on the source Dockerfile leads to a better quality of the resulting Docker image, specifically for its efficiency. The existing catalogs of best practices are focused on the quality of the code, more than the practical impact on the resulting Docker image. Chapter 5 aims to quantitatively measure the impact of the changes applied to improve performance, and the results empirically provide a set of recommendations of which changes developers should apply to make those improvements and improve the resulting Docker images.

2.3.2 Quality Features and Metrics

Azuma *et al.* [5] conducted a study where they categorize self-admitted technical debts (SATDs) in Dockerfiles. They manually annotated more than 300 comments to identify possible technical debts. As a result, about the 3% of comments in Dockerfile are SATDs. Additionally, they proposed a classification of the identified SATDs identifying five classes and eleven subclasses. Code debt and test debt are common SATDs in Dockerfiles, where 42% of them are Docker-specific. Moreover, not all the SATDs are related to code quality, but also to different non-functional aspects (*e.g.,* design, testing, and maintainability). A step towards is provided by the study of Ksontini *et al.* [58], which conducted an empirical study on refactoring operations and solved technical debts applied in open-source Docker projects. Among their findings, refactoring operations to

reduce image size of Dockerfiles are mainly associated with the evolution of them. Also, they defined 24 new Docker-specific refactorings and technical debts. The identified refactoring operations are not strictly linked to Docker best practices checked by the *hadolint* tool. Thus, a detailed analysis focused on how best practices violations (*i.e.,* smells) are fixed and what of them are relevant, is still missing. On the other hand, their findings are useful as a reference to implement refactoring strategies. An interesting research direction is to investigate in-depth each category defined in their taxonomy, such as the refactoring operations to reduce the image size and build time, to better understand how they are applied in practice.

Zhang *et al.* [142] performed an empirical study on the impact of the evolutionary trajectories of Dockerfiles. The evolutionary trajectories describe the frequency and type of modifications performed by the Dockerfile project maintainers. Then, through a regression analysis, the authors evaluate their impact on the quality and image build latency. The results show that different types of evolutionary categories have a different impact on quality. They also show that the evolutionary trajectories are correlated with the quantity of best practices violations occurring in Dockerfiles.

Ibrahim *et al.* [51] conducted an empirical study to evaluate the differences among Docker images hosted on DockerHub to support users to select the most suitable image to be adopted. Their results show that official images are more popular than community images. They show that community images are more resource-efficient than the studied software systems. Also, there are fewer security vulnerabilities than in their respective official images. They mainly based the conducted analysis on the popularity of Docker images, showing that aspects such *officiality* and presence of vulnerabilities have an impact on how popular a Docker image is. The missing piece is to understand if developers consider those aspects when choosing a Docker image over others, in terms of adoptions.

The presence of security vulnerabilities in Docker images is another important aspect related to quality. Zerouali *et al.* [137] conducted an empirical study on the relation between the outdatedness of Docker images and the presence of bugs and security vulnerabilities. They analyzed more than 7,000 official and community Docker images. The results show that more than half of the Docker images are

outdated, and all of them are affected by security vulnerabilities related to the base image operating system (*e.g.,* `debian`). However, keeping up-to-date Docker images is not enough to avoid security vulnerabilities, but it ensures avoiding the most severe ones. A successive work conducted by Opdebeeck *et al.* [83] investigated more in-depth the relation between child and parent images by constructing the inheritance network of Docker images in DockerHub. Since most of the Docker images are based on other images, software vulnerabilities and bugs can propagate from parent to child. On the same path, Shu *et al.* [109] measured in detail the propagation of vulnerabilities. They analyzed the security vulnerabilities of Docker images hosted on DockerHub, finding that, on average, both official and community images contain more than 180 vulnerabilities, on average.

Other studies investigated the risk in terms of security attacks that vulnerable Docker images can suffer. Haque *et al.* [40] investigated in-depth the exploitability and impact of base-image vulnerabilities on 261 base images, providing suggestions on how to deal with them. Another type of occurring vulnerabilities are those related to design aspects of the Docker platform or ad-hoc usage cases [73].

We can conclude that the occurrence of smells is not the only metric to measure the quality level of a Docker image. Moreover, plenty of alternatives are available in DockerHub providing the same requirements, however with different quality levels. In Chapter 6 we analyze in depth the relationship between the quality features that can describe a Docker image. In detail, we extracted the quality metrics and features proposed in the literature, to understand which of them are more relevant for developers when choosing a specific Docker image over others, providing the same requirements. The study is complementary with the results of Ibrahim *et al.* [51]. We propose a different perspective, more focused on the quality aspects, considering in addition the real usages of the analyzed Docker images.

2.4 Supporting Tools for Docker Development

Several tools are available to support developers during the development of Docker images and Dockerfiles. One of the most used are those checking for writing best practices, already discussed in the previous sections. The most popular

is *hadolint* [1], which checks for a set of rule violations based on the official Dockerfile writing guidelines [18]. Similar tools from the literature are *Binnacle* [44], *DRIVE* [146], and *Parfum* [23], also checking for writing best practices. A similar community-maintained alternative is `dockerfilelint`.[15] Sonarqube,[16] the popular platform to assess code quality, integrates a specific scanner for Dockerfiles.[17] It is able to check various sets of rules regarding code smells, security issues, and bugs.

Other tools, instead, assess the security of Docker images. The Docker platform offers a built-in vulnerability scanner on Docker images, *i.e., Docker Scout*,[18] that checks for Common Vulnerabilities and Exposures[19] (CVE). Unfortunately, the tool requires a paid plan for an unlimited number of scans. *Docker Scout* has been recently added to DockerHub, where it is possible to check for the presence of CVEs on the hosted Docker images. An example is the report for the official *ubuntu*.[20] It is worth saying that the tool reports also the *Software Billing of Materials (SBOM)* useful to understand the dependencies of the image and, eventually, the presence of unnecessary packages. Several paid enterprise tools are also available, such as *snyk*[21] which is able to detect and fix security issues directly on Docker images. Another famous platform is *Anchore*.[22]

On the other hand, there are a lot of popular and valid open-source alternatives. An example is *clair-scanner* tool,[23] which also performs checks for the presence of CVEs on Docker images. Also, *Dockle*[24] is another well-known linter for security issues specifically for Docker images. Another popular tool is *docker-bench-security* [22] is a tool that checks for various security best practices for the deployment of Docker applications in production environments.[25]

[15] https://github.com/replicatedhq/dockerfilelint
[16] https://www.sonarsource.com/products/sonarqube/
[17] https://rules.sonarsource.com/docker/
[18] https://docs.docker.com/scout/
[19] https://cve.mitre.org/
[20] https://hub.docker.com/layers/library/ubuntu/lunar/images/sha256-ea1285dffce8a938ef356908d1be741da594310c8dced79b870d66808cb12b0f?context=explore
[21] https://snyk.io/partners/docker/
[22] https://anchore.com/
[23] https://github.com/arminc/clair-scanner
[24] https://github.com/goodwithtech/dockle
[25] https://www.cisecurity.org/benchmark/docker

Furthermore, some tools allow performing reverse engineering on Docker images to extract the source code that created each layer. An example is *whaler* [85] which, besides the source code, extracts additional information such as the main user account, environment variables, and exposed secrets inside the Docker image (*i.e.,* sensitive information such as login credentials). In other cases, the tools help developers to optimize and reduce the image size. An example is the *SlimToolkit*,[26] which allows minimizing container images by up to 30x, improving the security of them. Basically, it selects only the necessary binaries and resources from the input container image, used to create a new one.

Several tools are specific to support developers in writing Dockerfiles. Specifically, they take as input the context (*i.e.,* the root folder of the project) and suggest a template Dockerfiles based on it. An example is the tool *starter*,[27] which generates a Dockerfile and a *docker-compose.yml* file from arbitrary source code. Some other tools provide more in-depth support for specific programming languages, based on the project context. For example, there are tools specific for R [82], Node.js,[28] Ruby,[29] and PHP [87]. Others support multiple languages, such as *generator-docker*.[30]

A more advanced support is provided by *GitHub Copilot* [36], which has been introduced as a general-purpose code completion tool but it works also for Dockerfiles. A similar approach is *Humpback* [39], but specifically designed for providing Dockerfiles completion. A different support i provided by the approach proposed by Zhang *et al.* [143], which helps developers choose the right *base image*. *ChatGpt*[31] has been evaluated on generative tasks related to software engineering [63, 38]. Therefore, it can suggest Dockerfiles based on a textual description of the requirements, even if it might require a manual intervention to fix the generated Dockerfile, or else to provide a more detailed description of the requirements. DockerizeMe [46] can automatically infer the environment dependencies starting from Python source code, without requiring inputs from developers. However, it only targets the execution dependencies for Python

[26] https://github.com/slimtoolkit/slim
[27] https://github.com/cloud66-oss/starter/commit/3b10a91
[28] https://github.com/tudvari/dockerfile-generator/commit/fd4582a
[29] https://github.com/elct9620/boxing/commit/4e3a200
[30] https://github.com/microsoft/generator-docker/commit/9bb74be
[31] https://openai.com/blog/chatgpt

code. Ye *et al.* [134] proposed DockerGen, an approach that uses knowledge graphs, built upon 220k Dockerfiles, for generating Dockerfiles for a specific software application. Starting from a target software, DockerGen infers all the dependencies required for the execution environment including the selection of a suitable base image. The support provided by the tool is only limited to dependency recommendation, thus it is not able to generate complete Dockerfiles.

Plenty of studies evaluated the applicability of deep learning techniques in the context of software engineering [126, 76, 122], which has proven to be effective in various coding tasks [79]. However, to the best of our knowledge, no previous work investigated the applicability of deep learning techniques for generating Dockerfiles. Nevertheless, none of the existing tools and approaches are specific to Docker and able to generate complete Dockerfiles starting from high-level requirements, or else they are limited to specific programming languages. Also, a recent study reports that developers perceive the creation of Dockerfiles and images as a time-consuming activity [96]. This motivates the investigation of advanced techniques (*i.e.,* deep learning) to support developers in this activity. To this aim, Chapter 7 reports an empirical evaluation of the applicability of deep learning techniques for generating Dockerfiles starting from high-level requirements.

CHAPTER 3

Not all Dockerfile Smells are the Same: An Empirical Evaluation of Writing Practices by Experts

Similarly to source code, Dockerfiles can be affected by bad design choices, known as Dockerfile smells, as we discussed in detail in Section 2.2. Specifically, their presence is an indicator of a poor code quality, as they can lead to reliability issues [13] and security vulnerabilities [9]. Several tools have been proposed in the literature to detect Dockerfile smells [44, 146, 9], where *hadolint* [1] is the reference detection tool used by both in practice and by researchers.

As we discussed previously, several studies in the literature investigated how to identify Dockerfile smells and how often they appear [13, 66, 27], replying on *hadolint* as the state-of-the-practice tool. Nevertheless, Lin *et al.* showed that developers are becoming more and more aware of Dockerfile best writing practices, as the number of Dockerfile smells is decreasing over time [66]. Eng *et al.* [27] showed that they are still widely diffused in open-source Dockerfiles, as a large majority of the developers who create Dockerfiles are not Dockerfile experts [44]. This brings up a natural question, still unanswered: *How do expert developers perceive the Dockerfile smells captured by hadolint?* As previously mentioned,

most Dockerfile smells are defined based on Docker guidelines. However, expert Dockerfile developers, who often find themselves working on Dockerfiles, could help both the research community and practitioners understand (i) which Dockerfile smells really represent bad practices, and (ii) whether there are commonly recognized bad practices missed by the available catalog of Dockerfile smells.

In this chapter, we aim at answering such a question. We first perform a mining-based study in which we mine the code written by expert Dockerfile developers to understand which Dockerfile smells have been more diffused in their history. To this aim, we consider 39,242 Dockerfiles directly maintained by developers from Docker. Second, we ran a survey with 37 expert Dockerfile developers. We showed each participant three Dockerfiles, each one only affected by one of the currently known Dockerfile smells, and ask them if they notice any problem in them. This allows us to understand if they are able to perceive the bad practice and if they can spot any other bad practice. Specifically, the respondents had to indicate (i) what lines are affected by a bad pattern, (ii) to what extent the pattern should be avoided, and (iii) what are the consequences of its presence. We obtained a total of 94 responses for 17 different smell categories selected from the 24 different smells occurring on Dockerfiles written by experts. Our results show that expert Dockerfile developers rarely perceive the Dockerfile smells in the currently available catalog as *relevant* bad practices, with three of them never spotted by any participant. Thus, we find that Dockerfile experts prioritize a part of the evaluated smells over others. More interestingly, Dockerfile experts indicated several other problems in the Dockerfiles we showed them. This allowed us to propose a taxonomy with 26 additional Dockerfile smells, which has been ranked by the relevance Dockerfile experts implicitly assign to each smell. Such a catalog can be useful for practitioners to have a wider picture of the typical bad practices when writing Dockerfiles.

The rest of this chapter is organized as follows. Section 3.1 presents the results of a mining study to understand if Dockerfiles written by experts adhere to best writing practices. In Section 3.2 we report a survey to understand how experienced developers perceive Dockerfile smells. In Section 3.3 we discuss the results, followed by the proposal of an enhanced catalog of Dockerfile smells in

Section 3.4. Section 3.5 presents the threats to validity. Finally, in Section 3.6 we summarize some final remarks along with future directions.

3.1 Study 1: Dockerfile Smells Affecting the Code Written by Experts

This first study has the *goal* of analyzing the Dockerfiles written by expert Dockerfile developers to understand whether they adhere to best practices and, if not, which of them they violate more frequently.

In detail, the study aims to address the following research question:

> *RQ_1: What are the best practice rules violated by Dockerfile experts?*

Our hypothesis is that expert developers perceive some best practices as not important and, thus, there is a higher number of violations for them.

3.1.1 Study Context

The context of this first study is composed of 39 official Dockerfile repositories, containing 39,242 unique Dockerfiles. The selected repositories come from the *Gold Set* dataset provided by Henkel *et al.* [44]. In detail, the repositories contain Dockerfiles created and maintained by experienced developers from Docker, *i.e.,* `docker-library`,[1], that are part of the *Docker official images program.*[2] In Henkel *et al.*'s study, they also compared the occurrence of best practice violations for Dockerfiles written by less-experienced developers with those from official repositories, which resulted to be worse. They reported the same on a set of Dockerfiles from industrial projects. We updated the list of the official repositories and retrieved the latest change history at the current time (*i.e.,* 2023-01-11), obtaining a total of 37,058 commits. The final dataset can be found in our replication package [2].

[1] `https://github.com/docker-library/`
[2] `https://github.com/docker-library/official-images\#what-are-official-images`

3.1.2 Experimental Procedure

The first step to answer RQ_1 was to extract all the Dockerfile snapshots in the change history of the 39 official Docker repositories. We discarded all the syntactically wrong Dockerfiles. Specifically, we discarded Dockerfiles for which either (i) *hadolint* returns DL1000 (invalid Dockerfile instructions), DL3061 (invalid instruction order), or DL3022 ("`COPY --from`" should reference a previously defined `FROM` alias) or (ii) the official Dockerfile parser[3] returns a parsing error. In the end, we excluded 724 Dockerfile snapshots.

In detail, we report the unique smells affecting all the snapshots over time for each Dockerfile. This means that, if a rule violation occurs in all the snapshots of the Dockerfile, we consider it as a single occurrence.

3.1.3 Empirical Study Results

We report in Table 3.1 the ranked list of Dockerfile smells identified by *hadolint* on the Dockerfiles in the official repositories maintained by Docker. The most occurring smells are DL4006 (use of `pipefail` for piped operations), DL3008 (missing version pinning for `apt-get`), DL3003 (Use `WORKDIR` to switch to a directory), and DL3047 (missing flag --progress for *wget*). The high occurrence of DL4006 could have two explanations. First, it could be related to the fact that developers do not care about such a smell. Second, it could be related to the fact that such a smell is not easy to spot and, thus, gets easily unnoticed. The high occurrence of DL3008 (18%) and DL3018 (11%), instead, more likely suggests that experts do not care about version pinning of OS packages since this smell is easier to notice. Indeed, it can be seen that this is not the same for software dependencies of *pip* (`DL3013`), and *gem* (`DL3028`), which instead are often pinned with versions and, thus, they present a lower occurrence of such rule violations (*i.e.,* less than 1% of relative occurrences). A reason could be that while developers find it necessary to pin libraries and framework versions for ensuring API compatibility with their source code, they find it less useful for OS dependencies that are probably considered less likely to cause such kind of problems. This is

[3] `https://github.com/asottile/dockerfile`

Table 3.1: Frequency of Dockerfile smells detected using *hadolint*.

Smell Type	**Description**	# **Occ.**
DL4006	Set `pipefail` to avoid silencing errors	1,550
DL3008	Pin versions in apt-get install	1,117
DL3003	Use WORKDIR to switch to a directory	1,014
DL3047	Avoid bloated logs using `--progress` with `wget`	774
DL3018	Pin versions in `apk add`	615
DL3059	Multiple consecutive `RUN` instructions	253
DL3015	Avoid additional packages by specifying `--no-install-recommends`	237
DL3019	Use the `--no-cache` flag with `apk`	176
DL3006	Always tag the version of a base image explicitly	90
DL3009	Delete the `apt-get` lists	43
DL3033	Pin version in `yum install`	43
DL3042	Avoid cache directory with `pip` using `--no-cache-dir`	32
DL3013	Pin versions in `pip`	18
DL4001	Use only `wget` or `curl`, not both	15
DL3020	Use `COPY` instead of `ADD` for files and folders	15
DL3028	Pin versions in `gem install`	13
DL3041	Pin version in `dnf install`	13
DL3038	Use the `-y` flag in `dnf install`	12
DL3014	Use the `-y` flag in `apt-get install`	6
DL3027	Prefer `apt-get` instead of `apt`	6
DL3007	Prefer an explicit version tag instead of `latest`	3
DL3025	Use arguments JSON notation for `CMD` and `ENTRYPOINT`	2
DL4000	`MAINTAINER` is deprecated	1
DL3016	Pin versions in `npm install`	0

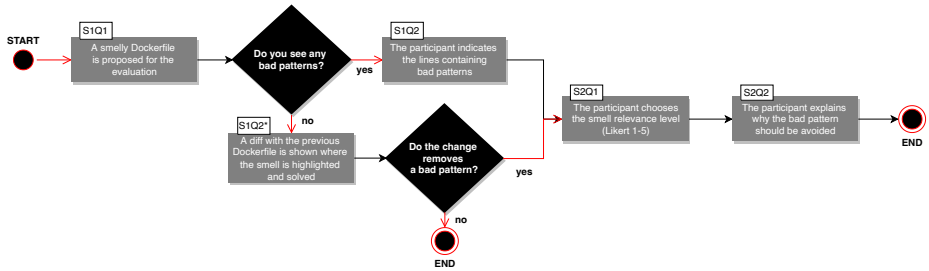


Figure 3.1: Summary workflow of the survey conducted to answer RQ_2. Each *action* represents a survey question, corresponding to an identifier. For example, S1Q2 identifies the first question (Q1) of section one (S1).

further confirmed by the fact that there are no occurrences for of `DL3016` (Pin versions in `npm`).

The less occurring smells are DL4000 (deprecated `MAINTAINER`), DL3025 (JSON notation for `CMD` and `ENTRYPOINT`), and DL3007 (using *latest* as base image tag). The only occurrence of DL4000 is related to a Dockerfile that kept the `MAINTAINER` instruction after its deprecation date, even if it was removed shortly after. However, DL4000 applies only after the effective deprecation of `MAINTAINER` in 2017. In total, there are only 2 occurrences of this smell, one before that date and another immediately after its deprecation. This is indicative that experts keep attention to the official Docker guidelines. Finally, it can be observed that developers refrain from using the `latest` tag for base images (DL3007, present in only three Dockerfiles). On the other hand, it often happens that they do not tag base images at all (DL3006, occurring in 90 Dockerfiles). Those two smells have the exact same implications because "latest" is automatically used when the base image tag is missing. However, "latest" is probably more noticeable, while a missing tag can more easily get unnoticed.

Summary of RQ_1: Some Dockerfile smells frequently occur even in repositories maintained by Dockerfile experts. DL4006 (use of `pipefail` for piped operations), DL3003 (Use `WORKDIR` to switch to a directory), along with missing version pinning (DL3008 and DL3018) are among the most occurring ones.

3.2 Study 2: Experts' Point of View

The *goal* of the second study is to understand how expert Dockerfile developers perceive Dockerfile smells, and if there are bad practices that are not captured by the current tools.

The second study aims to answer the following research question:

> *RQ_2: Are the Dockerfile smells considered bad practices by expert Dockerfile developers?*

We want to know if expert Dockerfile developers identify the presence of Dockerfile smells, considering them as bad writing practices that should be avoided. Thus, we want to gather feedback on the importance (or not) of the identified smells and bad practices.

To answer our research question, we ran a survey with expert Dockerfile developers from the open-source community. We explain below how we selected participants and smells, and how we designed the study.

3.2.1 Context Selection

Our context is composed by both *subjects* and *objects*. The *subjects* are expert Dockerfile developers, while the *objects* are Dockerfile smells. We describe below the procedure we used to select both subjects and objects for our study.

Participants Selection

Our target population is a subset of developers with extensive experience in Dockerfile development. It is particularly difficult to find developers with such characteristics in the open-source context as, in general, the Docker community has few expert developers compared to other developer communities [41]. We opted for a sampling strategy that consists of the selection of contributors from GitHub repositories, which is a common practice in the literature [147, 47, 65, 131, 42, 64]. In detail, we perform a *convenience sampling* [29] to select a subset of experienced Dockerfile developers from source repositories of the most popular public Docker images. To achieve this, we started from the set of repositories used

to run the first study and extended it by including the repositories of the most pulled community images from DockerHub, extracted from the dataset proposed by Lin *et al.* [66]. Such a dataset contains the metadata for ∼3M DockerHub images along with their source repository, for a total of ∼440k GitHub and BitBucket repositories. After ranking the Docker images by the number of pulls, we selected a subset of the corresponding GitHub repositories of the top-ranked images. Thus, we selected a total of 3,048 repositories, including 47 Docker official image repositories. From such a set of repositories, we extracted all the contributors that performed (i) at least 2 contributions, (ii) contributed to at least 2 different repositories, and (iii) had public contact information. As for the latter point, we discarded developers for which we had the email address extracted from the git logs but who did not publicly share it in their GitHub profile for privacy-related reasons. We selected, in the end, 931 potential participants.

We sent the invitations according to online survey best practices [35, 90]. Only 37 of them agreed to participate (∼4.0% acceptance rate). Both the absolute number of participants and the acceptance rate are comparable with similar studies [88, 124, 112, 65, 64].

Dockerfile Smells Selection

Instead of running our study on *all* the Dockerfile smells, we decided to do it on a representative subset of the smells, mainly by grouping similar smells and not considering those that we have not found in Study 1 when analyzing the official Dockerfile repositories. We started with the full set of Dockerfile smells. Then, we removed smells that never affected official Dockerfile repositories (as reported in the results of our first study). We were left with 24 Dockerfile smells. Note that we kept also a smell that never occurred in any official Dockerfile (DL3016, pin versions for `npm`) since it is very similar to other smells that actually occurred (*i.e.,* DL3013 and DL3028, which are related to different technologies, *i.e.,* pip and gem, respectively).

At this stage, we discarded the smells that are similar to others (*i.e.,* they are a variant for other tools or OSes) and that would reasonably be redundant for our study. Since *ubuntu* is the most popular base image, we excluded DL3018, DL3033, and DL3041, since they are variants of DL3008 (Missing version pin-

ning of `apt-get` packages) for different — less popular — OS package managers. Similarly, we discarded DL3019 because it is equivalent to DL3009 (deletion of `apt-get` sources lists) but for *apk*, and DL3038, which is equivalent to DL3014 (use the `-y` switch for `apt-get` install) but for the `dnf` package manager.

We discarded D3028 and DL3013 because they are variants of DL3016 (pin versions for `npm`) for `gem` and `pip`. Also, in this case, we selected the smell related to the more popular technology, based on the number of available packages.[4]

In the end, we were left with a total of 17 smells.

3.2.2 Experimental Procedure

We report below how we collected data and how we analyzed them to answer our research question.

Data Collection

We first prepared 17 Dockerfiles, one for each selected Dockerfile smell. The first author wrote such Dockerfiles, starting from those present in the open-source community (*i.e.,* git repositories and tutorials). The aim is to produce *clean* Dockerfiles (*i.e.,* without smells) that are as close as possible to the starting open-source example. Next, we ran *hadolint* on all of them to make sure that it detected no issues. Then, we artificially injected one of the smells on each Dockerfile. Again, we ran *hadolint* on all of them to make sure each of them had only the smell we decided to inject. In that case, the aim is to have only one smell per Dockerfile to avoid any bias that can come from the co-occurrence of multiple smells. Finally, we prepared 17 tasks for the participants, each of them regarding a randomly chosen Dockerfile among the ones we created. We detail the structure of the tasks below. Note that the final aim of each task is to evaluate if developers perceive the smell as bad pattern to avoid, and not to evaluate they skill in identifyng smells.

Participants could run the survey offline, whenever they preferred (*i.e.,* we did not have execution control). It was structured as follows:

[4] `npm` counted more than 2.1M packages in 2022 [81], while `pip` had ∼350k in the same year [89] and `gem` has 178k packages in 2023 [103].

Pre-questionnaire. The survey starts with a form in which there is (i) a description of the purpose of the survey, (ii) information about the data we collect, and (iii) a request for consent in which the participant agreed with the reported information. We also ask for general information about the professional experience: the current working position, total years of programming experience, experience with Docker development, and how much time they spend on open and closed-source projects. We request only the minimal information to assess the experience of the participants, as in some cases this kind of questions can discourage some of them from completing the survey [80].

Task execution. Next, participants were asked to complete a total of three tasks, randomly selected among the ones having the lowest number of already provided evaluations. We did this to keep a balanced number of evaluations for each Dockerfile smell. This is to obtain, at the end, approximately the same number of validations for each smell.

A summary workflow of each task is depicted in Fig. 3.1. In the first step (*i.e.,* S1), participants were initially presented with the Dockerfile at hand. They were asked whether they noticed any bad practices in the provided Dockerfile. It is worth noting that, here, we do not explicitly refer to "Dockerfile smells", but we ask participants to identify "bad practices". This is because we want to keep the concept of bad practices as open as possible so that developers can indicate what they consider a bad practice. If they did not find any bad practices, we presented them with the original version of the Dockerfile (*i.e.,* without the Dockerfile smell) and asked them if the performed change removed any bad practices. If the answer to the latter question was no, the survey ended. If, instead, the answer to the second question (or to the first one) was positive, participants were asked to report (in an open text box) any identified bad practice, along with the respective affected line numbers. In the second step (*i.e.,* S2), we asked to indicate to what extent they perceived each identified bad practice as relevant (Likert scale from 1 to 5). We also asked to specify why the identified bad practice should be avoided according to them.

Post-questionnaire. In the last part, participants could provide contact information and consent to reach them out later for being asked additional questions. The goal of this was to send them a very short *post-questionnaire* in which

Table 3.2: Survey questions asked to answer RQ2.

	ID	Question	Answer Type
Pre-survey	S0Q1	What is your current primary occupation?	Multiple choice
	S0Q2	What kind of projects do you spend most of your time on?	Multiple choice
	S0Q3	How many years of experience do you have in software development activities?	Multiple choice
	S0Q4	How long have you been using Docker and Dockerfiles during your development activities?	Multiple choice
	S0Q5	How do you estimate your expertise in writing Dockerfiles compared to the most experienced person you work/have worked with (1-5)?	Likert scale (1-5)
Section 1	S1Q1	Considering the proposed Dockerfile, do you think it is necessary to apply some improvements to the code (e.g., fix bad patterns) in order to work correctly and without problems in a production environment?	Multiple choice
	S1Q2	Please specify which lines you would improve and the reason for the improvement. For example, "Line X: Contains problem Y..."	Open-Ended
	S1Q2*	Here you can see the previous Dockerfile with some modifications applied. In your opinion, do these changes improve the Dockerfile by fixing a bad pattern?	Multiple choice
Section 2	S2Q1	To what extent do you think that the previously reported issues (i.e. bad writing patterns) must be avoided in a Dockerfile that has to be used in a production environment (1-5)?	Likert scale (1-5)
	S2Q2	Can you explain the answer provided in the question above? Please assume that you are talking to a novice developer who is learning to write Dockerfiles. Examples are: "You should avoid the pattern X as it can cause issues when..." "Using the pattern X is ok if you do not care about coding conventions..."	Open-Ended

we explicitly asked (i) if they know what Dockerfile smells are and (ii) if they use tools that support the quality assessment of Docker artifacts.

The survey requires about 15 minutes, and 5 minutes for the *post-questionnaire* that we sent later (*i.e.,* after a month, approximately). We report in Table 3.2 all the questions contained in the survey for each step. We ran a pilot study with 11 participants — personal contacts of the authors having different roles and experience in software development — to ensure that the questions were clear and to avoid any misleading interpretation. The survey was updated after each feedback was received.

We provide the Dockerfiles used in each task in our replication package [2].

3.2.3 Data Analysis

To answer RQ_2, we report a quantitative and qualitative analysis based on the responses obtained from our survey. We follow the common practices used for survey analysis [56, 64]. To answer RQ_2, we report some general information about the professional background of the participants (*i.e.,* from the *pre-questionnaire*

questions) to describe the demographics of the selected population. Then, we report what smells are considered bad practices by the experienced developers, along with the measure of how much these smells should be avoided (*i.e.,* assessed in *S2Q1*). Since the answers are open-ended, we needed to qualitatively analyze them to map the declared bad practices to the Dockerfile we were interested in. Simply, we manually verified if the provided open-ended question in which the respondents indicated the presence of a bad pattern (*i.e.,* S1Q2) corresponds to the smell under evaluation. For the cases in which the smell has been spotted in the second chance, there is no need for validation as the answer is a multiple choice. We counted in how many cases each Dockerfile smell was correctly identified. Also, we associated each identification to the step in which this happened (either *before* or *after* showing the participant the correct Dockerfile).

3.2.4 Results

We excluded two participants because they entered three invalid responses (*i.e.,* the open answers only contained dots or spaces). Thus, we obtained 94 valid responses collected iun a period of two weeks, appoximately. Most of the participants completed all the three tasks, while the others completed either one or two tasks and stopped the survey beforehand. On average, each participant completed 2.6 tasks. We first report some information about the demographics of the participants and, later, the details about the analysis we performed to answer RQ_2.

Demographics

When asked about their primary occupation, most of the participants reported that they are professional developers (35), except one who is a Master/PhD student. Half of the participants (17) mainly work on open-source projects, while 10 of them spend equal time on both open- and closed-source projects. Only 9 of the participants work mostly on closed-source projects. To effectively measure the overall programming experience and experience with Docker of the participants, we followed the guidelines provided by Siegmund *et al.* [110]. When asked about their overall programming experience, 23 of the responses are from those with

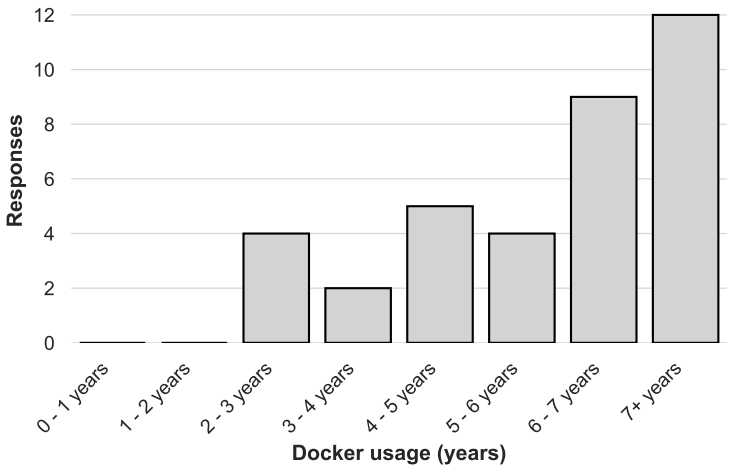


Figure 3.2: Docker experience (in years) of the survey participants.

more than 10 years of experience. Also, 5 participants have between 7 and 10 years of experience, and the same number between 5 and 7 years. The remaining responses (3) are from developers with between 1 and 5 years of experience.

In Fig. 3.2 we report a plot of the reported number of years of experience with Docker. 12 of the participants have more than 7 years of experience. This means that they have been using Docker almost since when it was introduced in 2013.

In Fig. 3.3 we report on the self-assessment of the expertise in Dockerfile writing compared to the most experienced present or past co-worker. Most of the participants identify themselves with a score of *4 - experienced* (15) or *5 - very experienced* (15). A total of 20 participants agreed to answer additional questions. After sending the invitation, 7 of them answered the questions of the *post-questionnaire* within a period of ten days, approximately.

RQ_2: Are the Dockerfile smells considered bad practices by expert Dockerfile developers?

In Table 3.3 we report the summary of the responses that we collected in our survey. Participants correctly identified the smell in 36% of the total cases. For 3 out of the 17 evaluated smells, however, they did not consider the smell

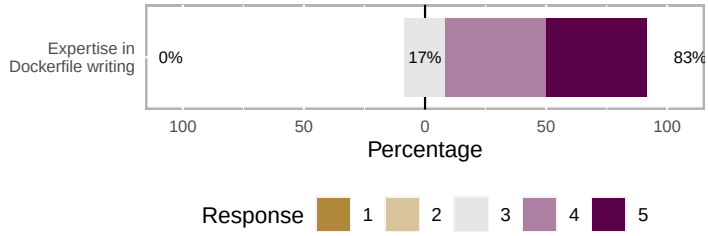


Figure 3.3: Dockerfile writing expertise measured using a Likert scale, varying from *1 - very inexperienced* to *5 - very experienced*.

Table 3.3: Number of Dockerfile smells identified by practitioners, with a representation of the identification percentage.

Dockerfile Smell	**# Identified**	**First Chance**	**Second Chance**
DL3007	4/5	4/4	0/4
DL3059	4/6	4/4	0/4
DL3006	4/6	3/4	1/4
DL3016	3/6	3/3	0/3
DL3014	2/4	2/2	0/2
DL4000	3/6	3/3	0/3
DL3042	3/6	2/3	1/3
DL3008	2/5	1/2	1/2
DL3025	2/6	2/2	0/2
DL4006	2/6	2/2	0/2
DL3027	2/6	0/2	2/2
DL3015	1/5	0/1	1/1
DL4001	1/5	0/1	1/1
DL3009	1/6	1/1	0/1
DL3003	0/5		
DL3047	0/5		
DL3020	0/6		

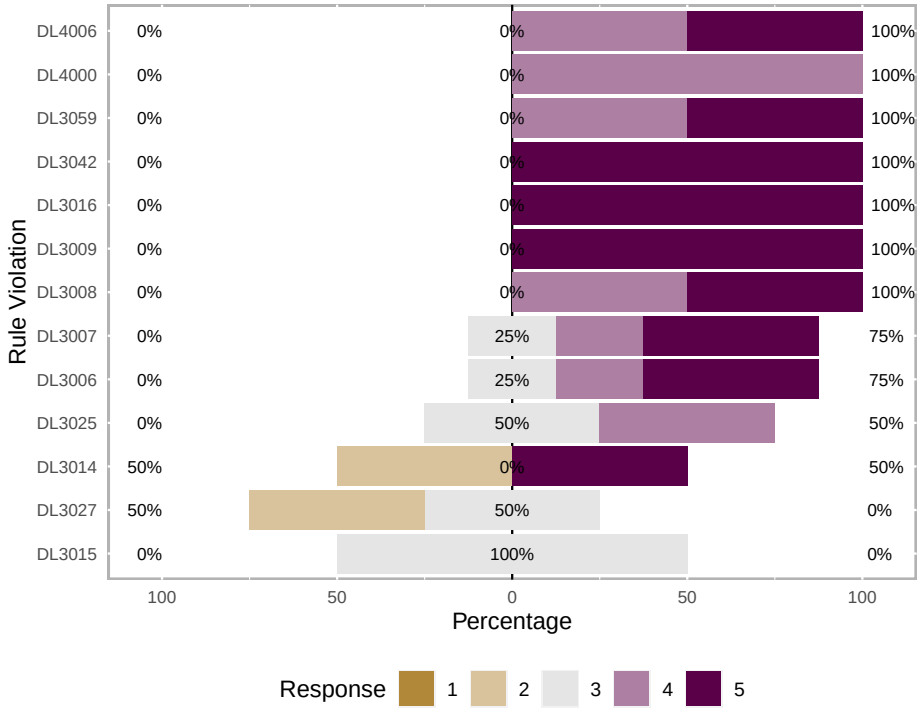


Figure 3.4: Developers' evaluation of the extent to which a Dockerfile smell should be avoided measured in S2Q1 (Table 3.2) using a 5-point Likert scale (from 1 - *"Strongly disagree"* to 5 - *"Strongly agree"*).

as a bad practice. Those are DL3003 (Use `WORKDIR` to switch to a directory), DL3020 (Use `COPY` instead of `ADD` for files and folders) and DL3047 (`wget` without flag `--progress`). Smells related to the base image version pinning, namely DL3006 and DL3007, received the highest percentage of identifications. The same is valid for DL3059 (multiple consecutive `RUN` instructions). Those smells have been identified in the first evaluation of the smelly Dockerfile requested in *S1Q1* (Table 3.2).

In Fig. 3.4 we report the smell relevance evaluated in *S2Q1*. Smells DL3009 (Deletion of `apt-get` sources lists), DL3016 (Pin versions for `npm`) and DL3042

(`-no-cache-dir` for `pip install`) have the highest level of agreement in terms of relevance (*i.e.,* all the evaluations "strongly agree" with the fact that the bad practice must be avoided). Interestingly, base image pinning smells (DL3006 and DL3007) received, in some cases, a neutral evaluation. This could be related to the fact that, when a specific version tag is missing, *latest* is used. Thus, developers interpret this as the latest version of that specific base image. For rule violations DL3027 (Prefer `apt-get` over `apt`) and DL3014 (Use the `-y` flag in `apt-get install`) some of the responses are "2 - *disagree*" or "3 - *neither agree nor disagree*". None of the evaluations expresses an agreement, thus we can conclude that those smells are likely to not be considered bad practices to avoid.

Moreover, looking at the results of the *post-questionnaire*, 5 of the 7 participants never heard about Dockerfile smells. When asking them about the quality issues they encountered in the past development experience, they reported: Too large base images, missing multi-stage builds, copying of unnecessary files, and layering issues. We also asked the respondents about supporting tools they know and use during development which are, in addition to *hadolint*: the Docker VS-Code extension, *shellcheck* (some of the violations are contained also in *hadolint*), the *docker inspect* command, *dive*[5] (*i.e.,* a tool to inspect Docker image layers), and *shfmt*[6] (*i.e.,* a shell script formatter).

Summary of RQ_2: Most of the smells are recognized by at least one participant, even though three of them are never identified. There is a difference among smells in terms of their identifiability and perceived relevance. In 64% of the cases, developers were not able to identify the Dockerfile smells.

3.3 Discussion

We distilled some lessons learned that will hopefully help both practitioners (for deciding their quality assurance policy on Dockerfiles), and researchers (for advancing the state-of-the-art on this topic and devising new approaches for detecting smells).

[5] `https://github.com/wagoodman/dive`
[6] `https://github.com/patrickvane/shfmt`

Lesson 1. Not all the Dockerfile smells are "smells". Comparing the results of the two RQs, it can be noticed that, generally, the most occurring smells are not perceived as such by practitioners. To confirm this intuition, we computed the Spearman's rank correlation coefficient between the two rankings (*i.e.,* most occurring smells and most perceived smells). We obtained a weak negative correlation ($\rho = 0.32$), *i.e.,* the more a smell occurs, the less it is perceived as such by practitioners. While the correlation is weak, this might still be an explanation for the presence of smells in official Dockerfiles: Some of them are simply not considered as such. This is particularly clear for some of them. For example, DL3003 (Use `WORKDIR` to switch to a directory), which occurs very frequently (1,014 times) in official Dockerfiles (RQ1), has never been perceived as such by expert developers (RQ2). Some other Dockerfile smells, instead, occur frequently even though they are perceived as such by expert developers. This is the case, for example, of DL3059 (multiple consecutive `RUN` instructions), which occurs in 253 official Dockerfiles even though developers perceive it as a bad practice (4/6 expert developers identified such a smell), and DL3006 (Missing version pinning for base image), which occurs in 90 official Dockerfiles even though it has been identified by 4 developers out of 6 in our survey. We believe that these Dockerfile smells depend on the context, and *hadolint* fails in catching this aspect. The following example reports a case we encountered of `FROM` instruction wrongly identified as a smell:

```
FROM alpine:{{env.variant | ltrimstr("alpine")}}
```

Specifically, it has been identified as smell DL3006 (Missing version tag) by *hadolint*. Since it selects the *alpine* version based on the value of the `variant` environment variable, the rule DL3006 is not violated because the variable in the Dockerfile will be replaced, before the build process, by a specific version tag of the base image. The community reported that, in some cases, knowing the context defines if a practice is "good" or "bad". An example is for smell DL3020.[7]

Another interesting case is smell DL3008. Pinning package versions requires a continuous maintenance to keep versions up-to-date as it could lead to reliability

[7] `https://github.com/hadolint/hadolint/issues/693`

issues in the future. This applies, specifically, to OS packages. Instead, missing version pinning for dependencies (*e.g.,* DL3013) is less diffused (Table 3.1).

Lesson 2. Developers prioritize performance and security. Looking at the overall topics reported in the survey responses, developers are mostly concerned about the image size, the build time and security issues. One of the participants explicitly reported that the effort invested in good writing practices should be focused on such aspects. An example smell is DL3020 (prefer `COPY` over `ADD` for files and folders), that has not been reported by none of the respondents. This could be related to the fact that, in most of the cases, it is not strictly related to the functionality of a Dockerfiles. In fact, for the analyzed Dockerfiles containing smell DL3020, the answers are about functional aspects (*e.g.,* avoid using *root* as container user). On the other hand, since the `ADD` instruction is fine when copying archives it is not wrong as a practice, compared to different cases (*e.g.,* using `MAINTAINER` which is deprecated [21]). *Hadolint*, however, fails in catching most of such aspects. This suggests that the actual smells do not fully cover all such implicit non-functional requirements of Dockerfiles. For example, only a few of them are for security best practices. In particular, rules DL3002 (Last user should not be `root`) and DL3011 (Invalid UNIX port range) are concerned with users and permissions. Furthermore, DL3059 (multiple consecutive `RUN` instructions) and DL3015 (Use `--no-install-recommends` to avoid installing additional packages) are some examples that can help to reduce the image size.

3.4 Towards an Enhanced Catalog of Dockerfile Smells

What we learned from the results of the two previous studies suggests that the current Dockerfile smells catalog is not comprehensive regarding bad practices. This is supported by the fact that when analyzing the responses of the survey participants in the second study (Section 3.2), we found that participants reported bad practices that are currently not part of the catalog of smells captured by *hadolint* (51% of the cases). In this section, we propose an enhanced catalog of smells, categorized in terms of their relevance, based on the obtained results. The

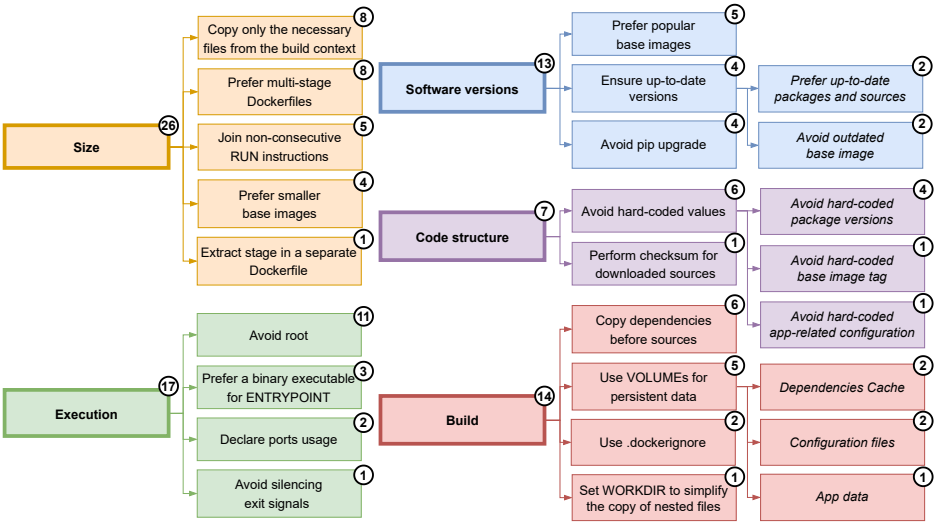


Figure 3.5: Categorization of the best practices recommendations provided by experts during our survey.

aim is to identify the basic best practices for which developers should care about when writing or maintaining Dockerfiles.

3.4.1 Collecting Recommendations from the Experts' Responses

As a first step, we wanted to categorize the best practices suggested by developers and not captured by *hadolint*. To do this, we performed a card-sorting-inspired approach [113] to categorize the new recommendations provided by the participants in the open-ended questions of our survey. First, one of the authors collected *as-is* all of those recommendations (*i.e.,* S1Q1 and S2Q2). Next, a second author validated those tags reaching a perfect agreement with the first author. After this, one of the authors proposed a first categorization of the selected recommendations, and subsequently discussed and re-arranged them together with a second author. The discussion phase has been repeated until reaching a consensus on the final categorization. The two authors excluded the recommendations that are not clear, or applicable only to the specific Dockerfile contect in which

they have been reported. We report in Fig. 3.5 a summary of the bad writing practices identified by the experienced developers indicating whether or not they correspond to an existing Dockerfile smell and why the smell should be avoided. We defined a total of five different macro-categories as described in the following.

Size. Being the category having the highest number of occurrences (26), it contains the recommendations that help to keep the size of the Docker image as small as possible. For example, preferring to use multi-stage Dockerfiles (8 occurrences) helps to keep the images small (*i.e.,* separating the build container from the execution container), and copying only the relevant files in the image from the build context (8 occurrences). This is, again, to keep the Docker image small avoiding unnecessary files (*e.g.,* "the pattern `COPY . .` should be avoided!").

Execution. With a total of 17 occurrences, this category contains the bad practices impacting the execution of the Docker image built from the Dockerfile. One of the most reported and important bad practices that is not captured in the current catalog is the usage of the default user `root` (11 occurrences). Dockerfile should change to a regular one to avoid security issues when running containers. Note that *hadolint* partially identifies such an issue (DL3002), but the tool is only able to detect explicit switches to root; if the user is *implicitly* root, it fails in identifying it. Also, developers should prefer a binary executable for `ENTRYPOINT` (3 occurrences). For example, a shell allows to easily debug containers. Another suggestion is to avoid silencing exit signals (1 occurrence). This is to avoid zombie processes (*i.e.,* orphaned containers) if the process exit signal is not handled correctly. An interesting suggestion consists in using the tool *tini* along with the starting command of the container to avoid the previously-mentioned issue.

Software versions. This category contains the recommendations to follow for the correct handling of the software and versions used in the Dockerfile (13 total occurrences). The most reported recommendation is to prefer popular Docker images (5 occurrences) because they are more likely to be maintained and updated. Also, it is important to ensure that both the packages and the base image versions are up-to-date (4 occurrences) to avoid reliability issues in the future due to outdated packages (*e.g.,* security vulnerabilities).

Build. With a total of 14 occurrences, this category contains the writing-best-practices to follow in order to improve the build process of a Docker image.

For example, copying and installing dependencies before sources (6 occurrences) allows to take advantage of the Docker cache mechanism, speeding up the successive builds by reusing the cached layers. Also, defining a `.dockerignore` file (2 occurrences) allows excluding unnecessary files from the build context when building images.

Code Structure. This category contains the recommendations to follow to improve the code readability and maintainability of the Dockerfile (7 occurrences). For example, a good practice is to avoid hard-coded values (6 occurrences) for the base image tag, software packages, and other non-static configurations (*e.g.,* ports) to make the Dockerfile code more flexible. Despite it being in contrast to version pinning smells, experts recommend using placeholders, along with default values, when specifying the version to easily maintain the Dockerfile and the resulting image up-to-date. In addition, it is better to perform a checksum of the downloaded sources (1 occurrence), *e.g.,* by using *wget*, to avoid corrupted files and security issues.

3.4.2 Ranking Dockerfile Smells

There are many developers with none or little expertise writing Dockerfiles that find themselves however in need of writing or maintaining a Dockerfile. These developers do not know on what aspects to focus on to have a good enough Dockerfile, in terms of writing quality.

As a further contribution of this chapter, based on the frequency of the bad practices identified by the developers, we propose a *ranked* list of the Dockerfile smells previously analyzed. In detail, we considered the frequencies of the best practices (i) identified in our second study (*i.e., hadolint* rules), and (ii) suggested by expert developers as "new" in their answers (described in Section 3.4.1). We perform min-max scaling for the frequency values of the two sets, independently. Then, we ordered them by the normalized frequency value. At the end, for each frequency value, we assign a rank.

In Table 3.4 we reported the ranked list of best practices. We also reported the overlap with other catalogs proposed in the literature, namely Binnacle [44], DRIVE [146], and DockerCleaner [9]. Thus, to meet a minimum quality level when writing Dockerfiles, developers should focus at least on the most frequently

Table 3.4: Ranked list of best practices along with the normalized frequencies. The icon represents whether the practice is suggested by experts (👤+) and/or included in existing catalogs, *i.e.,* hadolint (**H**), Binnacle [44] (🔭), DRIVE [146] (🖌), and Dockercleaner [9] (🤖).

Rank	Source	Description	Freq.
1	**H**	Prefer an explicit version tag instead of `latest` (DL3007)	**1.00**
1	👤+/**H**/🖌/🤖	Avoid root (~DL3002)	**1.00**
2	**H**	Always tag the version of a base image explicitly (DL3006)	**0.79**
2	**H**	Multiple consecutive `RUN` instructions (DL3059)	**0.79**
3	👤+/🖌	Prefer multi-stage Dockerfiles	**0.70**
3	👤+	Copy only the necessary files from the build context	**0.70**
4	**H**/🔭/🖌	Avoid cache directory with `pip` using `--no-cache-dir` (DL3042)	**0.52**
4	**H**/🔭/🖌	Use the `-y` flag in `apt-get install` (DL3014)	**0.52**
4	**H**	`MAINTAINER` is deprecated (DL4000)	**0.52**
4	**H**/🤖	Pin versions in `npm install` (DL3016)	**0.52**
5	👤+	Copy dependencies before sources	**0.50**
6	👤+	Prefer popular base images (official/community)	0.40
6	👤+	Join non-consecutive RUN instructions	0.40
7	**H**/🤖	Pin versions in apt-get install (DL3008)	0.37
8	👤+	Avoid hard-coded package versions	0.30
8	👤+	Avoid pip upgrade	0.30
8	👤+	Prefer smaller base images	0.30
9	**H**/🖌	Set `pipefail` to avoid silencing errors (DL4006)	0.25
9	**H**	Use arguments JSON notation for `CMD` and `ENTRYPOINT` (DL3025)	0.25
9	**H**	Prefer `apt-get` instead of `apt` (DL3027)	0.25
10	👤+	Prefer a binary executable for ENTRYPOINT	0.20
11	👤+	Declare ports usage	0.10
11	👤+	Use .dockerignore	0.10
11	👤+	Use VOLUME for Configuration Files	0.10
11	👤+	Use VOLUME for Dependencies Cache	0.10
11	👤+	Avoid outdated base image	0.10
11	👤+	Prefer up-to-date packages and sources	0.10
12	**H**/🔭/🖌/🤖	Use –no-install-recommends for `apt` (DL3015)	0.05
12	**H**	Use only `wget` or `curl`, not both (DL4001)	0.05
13	👤+	Extract stage in a separate Dockerfile	0.00
13	👤+	Avoid hard-coded base image tag	0.00
13	👤+	Avoid hard-coded app-related configuration	0.00
13	👤+	Use VOLUME for App Data	0.00
13	**H**/🔭/🖌	Delete the `apt-get` lists (DL3009)	0.00
13	👤+	Set WORKDIR to simplify the copy of nested files	0.00
13	👤+	Avoid silencing exit signals	0.00

reported best practices by experts (*e.g.,* ranks 1-5, normalized frequency ≥ 0.5). This means, for example, that they should pay attention to providing version pinning for the base image and dependencies (DL3006, DL3007, and DL3016), prefer using a regular user for Docker images, optimize the instruction order (*e.g.,* multi-stage build), and avoid the copy of unnecessary files (*e.g.,* copy only the required sources from the build context).

Moreover, some of those practices have been also discussed in the *gray literature*. For example, smell DL3007 (Version pinning for the base image) and "Avoid root" has been reported in a blog article about Docker best practices.[8] This means that a further investigation of the *gray literature* would help to build a more comprehensive catalog of the common practices suggested by developers.

3.5 Threats to validity

In this section, we report the threats to the validity of our study.

Construct Validity. The Dockerfile smells evaluated in our study are limited to those checked by *hadolint* which is currently the most popular tool adopted by both researchers and practitioners. The rules supported by the tool are mainly based on the official guidelines, which is not true for the other proposed tools [43]. We found customizations of *hadolint* in the official Dockerfiles evaluated in RQ_1, *i.e.,* the comment line `# hadolint ignore=DLXXXX` which disables the detection of one or more rules. This shows that, in addition to the popularity of the GitHub repository (*i.e.,* 9.1k stars), *hadolint* is adopted in practice to check violations in their Dockerfiles.

Internal Validity. The selection criteria of the survey participants could be perceived as not very strict (minimum of 2 contributions), for which we adopted a *convenience* sampling. We believe that the selected participants have a sufficient expertise level because (i) the Dockerfile developers community is smaller compared to others, and (ii) we selected a very specific population of those involved in repositories linked to some of the most popular Docker images available in DockerHub. Also, we relied on publicly available information, a common practice

[8] https://dev.to/techworld_with_nana/top-8-docker-best-practices-for-using-docker-in-production-1m39

used in similar studies [147], that misses closed source contributions. In addition, the survey participants could misunderstand the wording of some questions. To overcome this, we tested and adjusted the survey with 11 participants having different backgrounds (faculty, students, and developers) and are familiar with Docker. Finally, since we proposed ad-hoc defined Dockerfiles for our survey, they could not be representative of the overall population. However, they are inspired from open-source Dockerfiles to be as similar as possible to those used in practice.

External validity. In our survey, the participants identified bad writing patterns in the proposed Dockerfiles assuming that they have to be used in a production environment. This means that our findings might not be generalized to Dockerfiles written in different development contexts. Also, they are specific for Dockerfiles and the Docker platform. Finally, the bad practices not currently mapped by *hadolint* that participants could identify are those that we unintentionally introduced in the Dockerfiles proposed in the survey. It is very likely that more Dockerfile smells exist that are currently unknown.

3.6 Final Remarks

Docker is the leading technology for software containers, widely adopted in practice. Several best practices have been proposed and investigated in the literature, along with tools that support developers to avoid bad practices (*i.e.,* Dockerfile smells). This chapter first presents a study on official Dockerfiles to learn what smells appear most frequently in code written by *experts*. Also, we conducted a survey with expert Dockerfile developers to understand their perception of smells. We found that (i) official Dockerfiles contain smells, and (ii) expert Dockerfile developers perceive some of the smells as more important than others. As a final contribution, we defined a prioritized catalog of smells that provides a clear guide to less experienced developers to write better Dockerfiles.

To summarize, the takeaways from this chapter are the following:

- Some smells are more important than others. Expert developers prioritize the version pinning of base image and dependencies, the merge of consecutive `RUN` instructions, and other writing practices aimed at optimizing the

image size and the build time (*e.g.,* optimizing cache, multi-stage builds, and `.dockerignore` file);

- The survey participants reported additional bad practices, in 51% of the cases, that are currently not part of the existing catalog of smells. We can conclude that the current catalogs of writing recommendations should be extended to cover such aspects, *i.e.,* more focused on performance and security.

Therefore, future work in this direction should focus on the following aspects:

- The current catalog of Dockerfile smells could be extended by performing a systematic review of all the practices reported in both *gray* and *white* literature. We believe that in the future the community will converge on a common set of best practices for Dockerfiles;
- A missing point of the current catalogs is a lack of validation conducted by interviewing developers. This could be the next step of the work presented in this chapter, aimed at validating the proposed prioritized catalog of smells.

CHAPTER 4

An Empirical Study on Fixing Dockerfile Smells

As we already discussed, previous studies in the literature reported that Dockerfile smells are commonly diffused in open-source projects [13, 129, 66, 27], along with the diffusion of technical debt [5], and the refactoring operations typically performed by developers to optimize Dockerfiles [58].

While it is clear which Dockerfile smells are more frequent than others, it is still unclear which smells are more important to developers. A previous study by Eng *et al.* [27] reported how the number of smells evolves over time. Still, there is no clear evidence showing that (i) developers actually fix Dockerfile smells (*e.g.,* they might incidentally disappear), and that (ii) developers would be willing to fix Dockerfile smells in the first place.

In this chapter we propose an empirical study aimed at filling this gap. First, we analyze the survivability of Dockerfile smells to understand how developers fix them and which smells they consider relevant to remove. This, however, only tells a part of the story: Developers might not correct some smells because they are harder to fix. Therefore, we also evaluated to what extent developers are willing to accept fixes to smells when they are proposed to them (*e.g.,* by an automated

tool). The context of the study is represented by a total of ∼220k commits and 4,255 repositories, extracted from a state-of-the-art dataset containing the change history of about 9.4M unique Dockerfiles.

For each instance of such a dataset (which is a Dockerfile snapshot), we extracted the list of Dockerfile smells using the *hadolint* tool [1]. The tool performs a rule check on a parsed Abstract Syntax Tree (AST) representation of the input Dockerfile, based on the Docker [18] and shell script[1] best practices. Next, we manually validate a total of 1,000 commits that make one or more smells disappear to verify (i) that they are real fixes (*e.g.,* the smell was not removed incidentally), (ii) whether the fix is *informed* (*e.g.,* if developers explicitly mention such an operation in the commit message), and (iii) remove possible false positives identified by *hadolint*.

Then, we evaluated to what extent developers are willing to accept changes aimed at fixing smells. To this aim, we defined DOCKLEANER, a rule-based refactoring tool that automatically fixes the 12 most frequent Dockerfile smells. We used DOCKLEANER to fix a set of smelly Dockerfiles extracted from the most active repositories. Next, we submitted a total of 157 pull requests to developers containing the fixes, one for each repository. We monitored the status of the pull requests for more than 7 months (*i.e.,* 218 days). In the end, we evaluated how many of them get accepted for each smell type and the developers' reactions. The results show that, mostly, smells are fixed either very shortly (36% of the cases). There are also cases in which they are fixed after a very long period (2%, after 2 years). This could be a consequence of the fact that, generally, a few changes are performed on Dockerfiles and there the probability of noticing the errors is higher in the short-term (*e.g.,* until the Dockerfile works correctly) or, instead, it naturally increases with time, but very slowly. Also, developers perform changes on Dockerfiles mainly to optimize the build time and reduce the final image size, while there are only few changes limited only to the improvement of code quality. Even if Dockerfile smells are commonly diffused among Dockerfiles, developers are gradually becoming aware of the writing best practices for Dockerfiles. For example, avoiding the usage of `MAINTAINER` which is deprecated, or they prefer to use `COPY` instead of `ADD` for copying files and folders as it is suggested by

[1] `https://github.com/koalaman/shellcheck`

the Docker guidelines.[2] In addition, developers are open to approve changes aimed at fixing smells for the most common violations, but with some exceptions. Examples are the missing version pinning for `apt-get` packages (DL3008), which has received negative reactions from developers. However, version pinning, in general, is considered fundamental for other aspects, such as the base image pinning (DL3006 and DL3007), or the pinning of software dependencies (*e.g.,* `npm` and `pip`). To summarize, the contributions that we provide in this chapter are the following:

1. We perform a detailed analysis of the survivability of Dockerfile smells and manually validated a sample of smell-fixing commits for Dockerfile smells;

2. We introduce DOCKLEANER, a rule-based tool to fix the most common Dockerfile smells;

3. We run an evaluation via pull requests of the willingness of developers of accepting changes aimed at fixing Dockerfile smells.

The remaining of the chapter is organized as follows: Section 4.1 describes the design of our study, while in Section 4.3 we present the results of our experiment. In section Section 4.4 we qualitatively discuss the results. Finally, in Section 4.5 we discuss the threats to validity and in Section 4.6 we summarize the final remarks of this chapter and provide future directions.

4.1 Study Design

The *goal* of our study is to understand whether developers are interested in fixing Dockerfile smells. The *perspective* is of researchers interested in improving Dockerfile quality. The *context* consists in 53,456 Dockerfile snapshots, extracted from 4,255 repositories.

In detail, the study aims to address the following research questions:

[2] `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/\#add-or-copy`

RQ_1: *How do developers fix Dockerfile smells?* We want to conduct a comprehensive analysis of the survivability of Dockerfile smells. Thus, we investigate what smells are fixed by developers and how.

RQ_2: *Which Dockerfile smells are developers willing to address?* We want to understand if developers would find beneficial changes aimed at fixing Dockerfile smells (*e.g.,* generated by an automated refactoring tool).

4.1.1 Study Context

The context of our study is represented by a subset of the dataset introduced by Eng *et al.* [27]. The dataset consists in about 9.4 million Dockerfiles, in a period spanning from 2013 to 2020. To the best of our knowledge, the dataset is the largest and the most recent one from those available in the literature [13, 44, 58]. Moreover, such a dataset contains the change history (*i.e.,* commits) of each Dockerfile. This characteristic allows us to evaluate the survivability of code smells (RQ_1). The authors constructed that dataset through mining software repositories from the S version of the WoC (World of Code) dataset [72].

4.1.2 Data Collection

To avoid toy projects, we selected only the repositories having at least 10 stars for a total of 4,255 repos, excluding forks. We also discarded the repositories where the star number is not available in the original dataset (*i.e.,* the value is reported as `NULL`). We cloned all the available repositories from the selected sample to obtain the most updated commit data at the time our analysis started (*i.e.,* March 2023). Next, using a heuristic approach, we (i) identified all the Dockerfiles at the latest commit, and (ii) we traversed the commit history to get all the commits and snapshots for the identified Dockerfile. In detail, for the first step, we processed all the source files contained in the repository and we evaluated if the file (i) contains the word `"dockerfile"` in the filename, and (ii) if contains valid and non-empty commands, *i.e.,* can be correctly parsed using the official *dockerfile parser*.[3] For each valid Dockerfile, we mined the change

[3] `https://github.com/asottile/dockerfile`

history using `git log`. We excluded the Dockerfiles having only one snapshot (*i.e.,* no changes, referenced by only one commit). After this, we extracted a total of ∼220k commits corresponding to 53,456 unique Dockerfiles. In the end, we ran the latest version of *hadolint*[4] for each Dockerfile to extract the Dockerfile smells, if present.

4.2 Experimental Procedure

In this section, we describe the experimentation procedure that we will use to answer our RQs. Fig. 4.1 describes the overall workflow of the study.

4.2.1 RQ_1: *How do developers fix Dockerfile smells?*

To answer RQ_1, we perform an empirical analysis on Dockerfile smell survivability. For each Dockerfile d, associated with the respective repository from GitHub, we consider its snapshots over time, $d_1, \ldots, d_n$, associated with the respective commit IDs in which they were introduced (*i.e.,* $c(d_1), \ldots, c(d_n)$). Additionally, we consider the Dockerfile smells detected with *hadolint*, indicated as $\eta(d_1), \ldots, \eta(d_n)$. For each snapshot d_i (with $i > 1$) of each Dockerfile d, we compute the disappeared smells as $\delta(d_i) = \eta(d_i) - \eta(d_{i-1})$. All the snapshots for which $\delta(d_i)$ is not an empty set are *candidate* changes that aim at fixing the smells. We define a set of all such snapshot as $PF = \{d_i : |\delta(d_i)| > 0\}$. In the end, we obtain a set of smelly (d_{i-1}) and smell-removing commit (d_i) pairs. We implemented the described procedure as a basic heuristic approach, which (i) went through all the commits, (ii) executed *hadolint* to detect smells, (iii) returned the smelly and smell-removing commits pairs. The total time required was about nine hours.

Next, we manually evaluate the commit pairs to verify (i) that the changes that led to the snapshots in PF are actual fixes for the Dockerfile smell, and (ii) whether developers were aware of the smell when they made the change, and (iii) avoid any bias related to the presence of false positives in terms of smells (identified by *hadolint*). In detail, we manually inspect a sample of 1,000 of such

[4]hadolint release v2.12.0

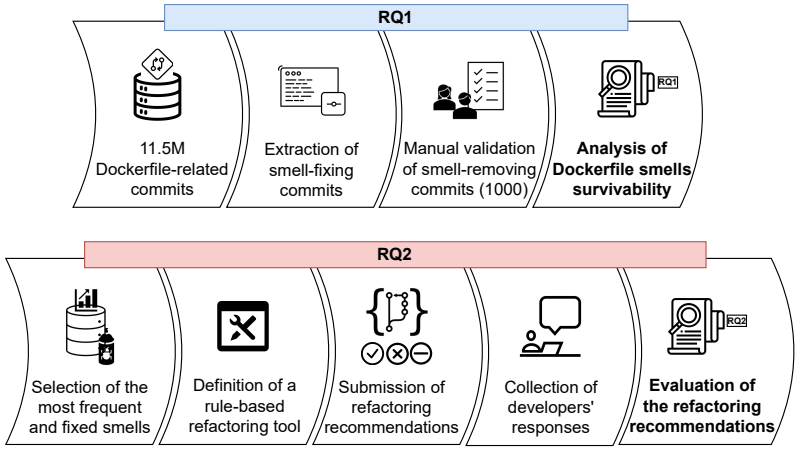


Figure 4.1: Overall workflow of the experimentation procedure.

```
@@ -2,5 +2,4 @@ FROM ubuntu:20.04
2  2
3  3    RUN apt-get install -y \
4  4        curl=7.* \
5  -        wget \
6  -        && rm -rf /var/lib/apt/lists/*
   5  +        && rm -rf /var/lib/apt/lists/*
```

Figure 4.2: Example of a candidate smell-fixing commit that does not actually fix the smell.

candidate changes, which is statistically representative, leading to a margin of error of 3.1% (95% confidence interval) assuming an infinitely large population. We look at the code diff to understand *how* the change was made (*i.e.,* if it fixed the smell or if the smell disappeared incidentally). Also, for actual fixes, we consider the commit message, the possible issues referenced in it, and the pull requests to which they possibly belong to understand the purpose of the change (*i.e.,* if the fix was informed or not). We identify as *smell fixing change* a commit in which developers (i) modified one or more Dockerfile lines that contained one or more smells in the previous snapshot (*i.e.,* commit), and (ii) kept the functionality expressed in those lines. For example, if the commit removes the instruction line where the smell is present, we do not label it as an actual smell-fixing commit. This is because the smelly line is just removed and not fixed (*i.e.,* the functionality changed). Let us consider the example in Fig. 4.2: The package `wget` lacks version pinning (left). An actual fix would consist of the addition of a version to the package. Instead, in the commit, the package gets simply removed (*e.g.,* because it is not necessary). Therefore, we do not consider such a change as a fixing change. Besides, we mark a fix as *informed* if the commit message, the possibly related pull request, or the issue possibly fixed with the commit explicitly reports that the modification aimed to fix a bad practice.

Two of the authors independently evaluated each instance. The evaluators discussed conflicts for both the aspects evaluated aiming at reaching a consensus. The agreement between the two annotators is measured using the Cohen's Kappa Coefficient [15], obtaining a value of $k = 0.79$ considered "*very good*" according to the interpretation recommendations [94]. The total effort required for the manual validation was about five working days, considering two of the authors that performed the annotation and discussed the conflicts.

Moreover, starting from the smell-fixing change, we go back through the change history to identify the *last-smell-introducing* commit, *i.e.,* the commit in which the artifact can be considered smelly [120], by executing `git blame` on the Dockerfile line number labeled as smelly by *hadolint*. In the end, we summarize the total number of fix commits and the percentage of actual fix commits. Moreover, for each rule violation, we report the trend of smell occurrences and

Table 4.1: The most frequent Dockerfile smells identified in literature [27], along with the most fixed rules we identified in our study (reported with ∗). We implemented all of the rules in DOCKLEANER.

Rule	Description	How to fix
DL3003	Use `WORKDIR` to switch to a directory	Replace `cd` command with `WORKDIR`
DL3006	Missing version pinning for base image	Pin the version tag corresponding to the resulting image digest
DL3008	Missing version pinning of `apt-get` packages	Pin the latest suitable package version from Launchpad
DL3009	Delete the `apt-get` lists after installing packages	Add in the corresponding instruction block the lines to clean `apt` cache
DL3015	Avoid additional packages by specifying `-no-install-recommends`	Add the option `-no-install-recommends` to the corresponding instruction block
DL3020	Use `COPY` instead of `ADD` for files and folders	Replace `ADD` instruction with `COPY` when copying files and folders
DL4000	`MAINTAINER` is deprecated	Replace maintainer with the equivalent `LABEL` instruction
DL4006	Set `-o pipefall` to avoid silencing errors in `RUN` instructions having pipe operations	Add the `SHELL` pipefail instruction before `RUN` that uses pipe
DL3059*	Consider consolidation for multiple consecutive `RUN` instructions	Concatenate all subsequent `RUN` instruction until a comment line or a different instruction
DL3007*	Avoid to use the latest to tag the version of an image	Same approach as DL3006
DL3025*	Use arguments JSON notation for `CMD` and `ENTRYPOINT`	Refactor the instruction command as JSON notation
DL3048*	Invalid Label Key	Refactor the `LABEL` instructions according to the *hadolint* documentation examples[5]

fixes over time, along with a summary table that describes the most fixed smells. We also discuss interesting cases of smell-fixing commits.

4.2.2 RQ_2: *Which Dockerfile smells are developers willing to address?*

To answer RQ_2, we first defined a list of rules, based both on the literature and the results of RQ_1, and then implemented a rule-based refactoring tool, DOCKLEANER, to automatically fix them. We defined the fixing rules as described in the *hadolint* documentation.[6] Next, we use DOCKLEANER to fix smells in existing Dockerfiles from open-source projects and submit the changes to the

[6] `https://github.com/hadolint/hadolint/wiki`

developers through pull requests to understand if they agree with the fixes and are keen to accept them. We describe these steps in the following sections.

Fixing rules for Dockerfile Smells

As a preliminary step, we identified a set of Dockerfile smells that we wanted to fix, considering the list of the most occurring Dockerfile smells, ordered by prevalence, according to the most recent paper on this topic [27]. However, we excluded and added some rule violations. Specifically, among the missing version pinning violations, we excluded DL3013 (*Pin versions in pip*) and DL3018 (*Pin versions in apk add*) because they are less occurring variants (*i.e.,* ∼4% and ∼5%, respectively) of the more prevalent smell DL3008 (15%), even if concerning different package managers. Additionally, we include in DOCKLEANER the most occurring smells resulting from the analysis performed in RQ_1 and not reported in the literature. We report in Table 4.1 the full list of smells target in our study, along with the rule we use to automatically produce a fix. It is clear that most of the smells are trivial to fix. For example, to fix the violation DL3020, it is just necessary to replace the instruction `ADD` with `COPY` for files and folders. In the case of the *version pinning*-related smells (*i.e.,* DL3006 and DL3008), instead, a more sophisticated fixing procedure is required. We refer to *version pinning*-related smells as to the smells related to missing versioning of dependencies and packages. Such smells can have an impact on the reproducibility of the build since different versions might be used if the build occurs at different times, leading to different execution environments for the application. For example, when the version tag is missing from the `FROM` instruction of a Dockerfile (*i.e.,* DL3006), the most recent image having the latest tag is automatically selected. To fix such smells, we use a two-step approach: (i) we identify the correct versions to pin for each artifact (*e.g.,* each package), and (ii) we insert the selected versions to the corresponding instruction lines in the Dockerfile. We describe below in more detail the procedure we defined for each smell.

Image version tag (DL3006). This rule violation identifies a Dockerfile where the base image used in the `FROM` instruction is not pinned with an explicit tag. In this case, we use a fixing strategy that is inspired by the approach of Kitajima *et al.* [54]. Specifically, to determine the correct image tag, we use the

`FROM ubuntu`	`FROM ubuntu:20.04`
(A) Smelly line	(B) Possible solution.

Figure 4.3: Example of rule DL3006.

image name together with the image *digest*. Docker images are labeled with one or more *tags*, mainly assigned by developers, identifying a specific version of the image when *pulled* from DockerHub. On the other hand, the *digest* is a hash value that uniquely identifies a Docker image having a specific composition of dependencies and configurations, automatically created at build time. The *digest* of existing images can be obtained via the DockerHub APIs.[7] Thus, the only way to uniquely identify an image is using the *digest*. To fix the smell, we obtain (i) the *digest* of the input Docker image through build, (ii) we find the corresponding image and its tags using the DockerHub APIs, and (iii) we pick the most recent tag assigned, that is different from the *"latest"* tag. An example of smell fixed through this rule is reported in Fig. 4.3.

Pin versions in package manager (DL3008). The version pinning smell also affects package managers for software dependencies and packages (*e.g.,* `apt`, `apk`, `pip`). In that case, differently from the base image, the package version must be searched in the source repository of the installed packages. The smell regards the *apt* package manager, *i.e.,* it might affect only the Debian-based Docker images. For the fix, we consider only the Ubuntu-based images since (i) we needed to select a specific distribution to handle versions (more on this later), and (ii) Ubuntu is the most widespread derivative of Debian in Docker images [27]. The strategy we use to solve DL3008 works as follows: First, a parser finds the instruction lines where there is the `apt` command, and it collects all the packages that need to be pinned. Next, for each package, the current latest version number is selected considering the OS *distribution* (*e.g.,* Ubuntu, Xubuntu, etc.), and the *distro* series (*e.g.,* 20.04 *Focal Fossa* or 14.04 *Trusty Tahr*). The series of the OS is particularly important, because they may offer different versions for the same package. For instance, if we consider the `curl`

[7] `https://docs.docker.com/docker-hub/api/latest/`

`RUN apt-get install -y curl`	`RUN apt-get install -y curl=7.*`
(A) Smelly line	(B) Possible solution.

Figure 4.4: Example of rule DL3008.

package, we can have the version `7.68.0-1ubuntu2.5` for the *Focal Fossa* series of Ubuntu, while for the series *Trusty Tahr* it equals to `7.35.0-1ubuntu2.20`. So, if we try to use the first in a Dockerfile using the Trusty Tahr series, the build most probably fails. The final step consists in testing the chosen package version. Generally, a package version adopts semantic versioning, characterized by a sequence of numbers in the format <MAJOR>.<MINOR>.<PATCH>. However, the specific versions of the packages might disappear in time from the Ubuntu central repository, thus leading to errors while installing them. Given that the `PATCH` release does not drastically change the functionalities of the package and that old patches frequently disappear, we replace it with the symbol '*', indicating "any version," in such a way the *latest* version is automatically selected. After that, a simulation of the `apt-get install` command with the pinned version is executed to verify that the selected package version is available. If it is, the package can be pinned with that version; otherwise, also the `MINOR` part of the version is replaced with the '*' symbol. If the package can still not be retrieved, we do not pin the package, *i.e.,* we do not fix the smell. Pinning a different `MAJOR` version, indeed, could introduce compatibility issues and the developer should be fully aware of this change. An example of a fix generated through this strategy is reported in Fig. 4.4. It is worth saying that we apply our fixing heuristic only to packages having missing version pinning. This means that we do not update packages pinned with another version (*e.g.,* older than the reference date used to fix the smell). Moreover, in some cases, developers might not want the pinned package version, but rather a different one, despite the version we pin is most likely the closest one to the one they originally tested their Dockerfile on. For example, they want a newer version of that package (*e.g.,* the latest). We discuss those cases during the evaluation phase of the automated fixes via pull requests.

Evaluation of Automated Fixes

To evaluate if the fixes generated by DOCKLEANER are helpful, we propose them to developers by submitting the patches on GitHub via pull requests. The first step is to select the most active repositories to ensure responses for our pull requests. To achieve this, we select a subset of repositories from our study context ensuring that, each repository, (i) contains at least one Dockerfile affected by one or more smells that we can fix automatically (reported in Table 4.1), and (ii) at least one pull request merged, along with commit activity, in the last three months. In this way, we select a total of 186 repositories containing 829 unique Dockerfiles affected by 5,403 smells. The next step is to associate each repository with a specific smell corresponding to a single Dockerfile to fix. This is to avoid flooding developers with pull requests.

We used a greedy algorithm to select the smell to fix in the Dockerfiles from the candidate repositories to ensure each of them is considered a balanced number of times. We start from the less occurring smells among all the available repositories, and we iteratively (i) select one target smell to fix, (ii) randomly select one Dockerfile candidate containing that smell, (iii) assign the repository to that smell to mark it as unavailable for the successive iterations, and (iv) increment a counter, for each smell, of the assigned Dockerfile candidates. The algorithm stops when there are no more repositories available. The counter of assigned smells is used, along with the overall smell occurrence, in the first step of the heuristic. This ensures that, for each iteration, we consider the smell (i) having the lower occurrence and (ii) is currently assigned for the fix to a lower number of repositories. In this phase we manually discard smells that can not be fixed by DOCKLEANER. For example, for DL3008, we only support Ubuntu-based Dockerfiles, but the smell might also affect the Debian-based ones. In total, we excluded 14 smells.

At the end of that procedure, we followed the commonly used `git` workflow best practices for opening the pull requests. Specifically, we first created a fork for the target repository. Then, we created a branch where the name follows the format `fix/dockerfile-smell-DLXXXX`. Finally, we signed-off the patches as it is required by some repositories (as well as being a good practice), and we submitted the pull request.

> Hi,
> The Dockerfile placed at {`dockerfile_path`} contains the best practice violation {`violation_id`} detected by the *hadolint* tool.
> The smell {`violation_id`} occurs when {`violation_description`}
>
> This pull request proposes a fix for that smell generated by my fixing tool. The patch was manually verified before opening the pull request. To fix this smell, specifically, {`fixing_rule_explanation`}.
>
> This change is only aimed at fixing that specific smell. If the fix is not valid or useful, please briefly indicate the reason and suggestions for possible improvements.
>
> Thanks in advance.

Figure 4.5: Example of the pull request message. The placeholders (wrapped in curly braces) will be replaced with the corresponding values.

To do this, we defined and used a structured template for all the pull requests, as reported in Fig. 4.5. We manually modified the template in the cases where the repository requires a custom-defined guidelines. The time required by DOCKLEANER to generate the fixing recommendations is only a few seconds for the simpler fixing procedures (*e.g.,* replacing `COPY` with `ADD`). For the more complex ones, such as version pinning, it can even take a few minutes.

For the evaluation, we adopted a methodology similar to the one used by Vassallo *et al.* [125]. In detail, we monitored the status of each pull request for more than 7 months (*i.e.,* 218 days, starting from the last created pull request date) to allow developers to evaluate it and give a response. We interacted with them if they asked questions or requested additional information, but we did not make modifications to the source code of the proposed fix unless they are strictly related to the smell (*e.g.,* the fixing procedure of the smell is reported as not valid). We report such cases in the discussion section. At the end of the monitoring period, we tagged each pull request with one of the following states:

- *Ignored*: The pull request does not receive a response;

- *Rejected/Closed*: The pull request has been closed or is explicitly rejected;
- *Pending*: The pull request has been discussed but is still open;
- *Accepted*: The pull request is accepted to be merged but is not merged yet;
- *Merged*: The proposed fix is in the main branch.

For each type of fixed smell, we report the number and percentage of the fix recommendations accepted and rejected, along with the rationale in case of rejection and the response time. Also, we conducted a qualitative analysis of the developers' interactions. In particular, we analyzed those where the pull request is rejected or pending to understand why the fix was not accepted. For example, the fix might have been accepted because the developers were not interested in performing that modification to their Dockerfile. Moreover, we analyze the additional information that the developer submits on rejected pull requests, from which we extract takeaways useful for both practitioners and researchers. Using a card-sorting-inspired approach [113] performed by two of the authors on the obtained responses, we identified a set of categories that we used to classify the developers' reactions to rejected pull requests.

Data Availability

The code and data used in our study, along with the implementation of DockLeaner, can be found in the replication package [102].

4.3 Analysis of the Results

In this section, we report the analysis of the results achieved in our study to answer our research questions.

4.3.1 RQ_1: *How do developers fix Dockerfile smells?*

We report in Fig. 4.6 the trend of the 10 most occurring Dockerfile smells among the Dockerfile snapshots we analyzed. To plot this figure, we collected all the unique Dockerfiles (based on their path and repository) for each year, then

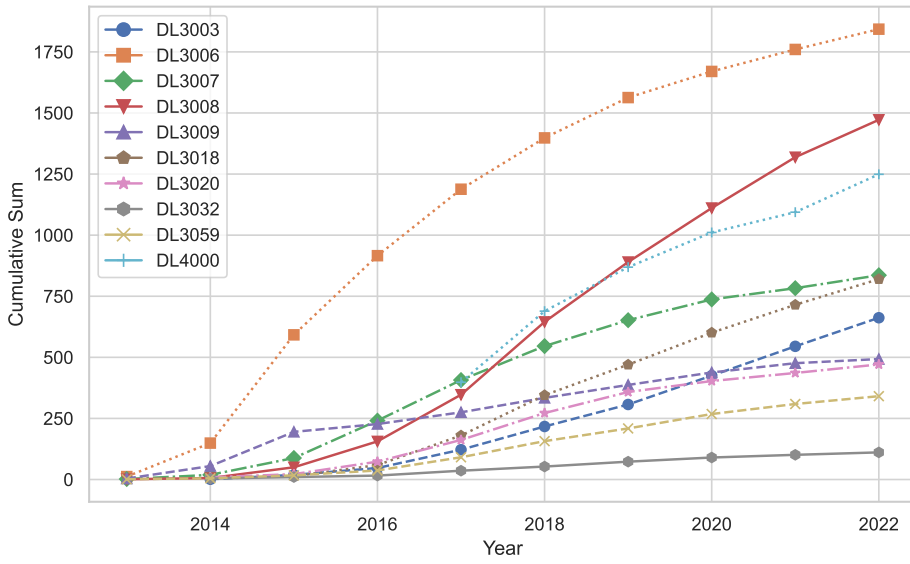


Figure 4.6: Occurrence over time for the top 10 Dockerfile smells.

we extracted and counted all the smells of the latest version of each of them (for each year).

The most occurring smell is DL3006 — version pinning for the base image— , followed by DL3008 — missing version pinning for `apt-get`—, which is also the most growing one, and DL4000 — deprecated `MAINTAINER`—. Since smell DL4000 became a bad practice in 2017[8] due to the deprecation of the `MAINTAINER` instruction, we excluded its occurrences before that date from the plot.

In our manual validation, we found that 33.6% of the commits in which smells disappear actually fix smells. We report in Table 4.2 a summary of the characteristics of such commits for the smells for which we found at least 5 fixes (from a total of 572 fixed smells). In detail, we report the total number of fixing commits, and the average fixing time, measured both as days and the number of commits that elapsed between the last commit introducing a smell and the smell-fixing commit. Additionally, we report in Fig. 4.8 the adjusted boxplots describing the days that passed after each smell got fixed. We report in Fig. 4.7 the fixing trend

[8] `https://docs.docker.com/engine/release-notes/prior-releases/#1130-2017-01-18`

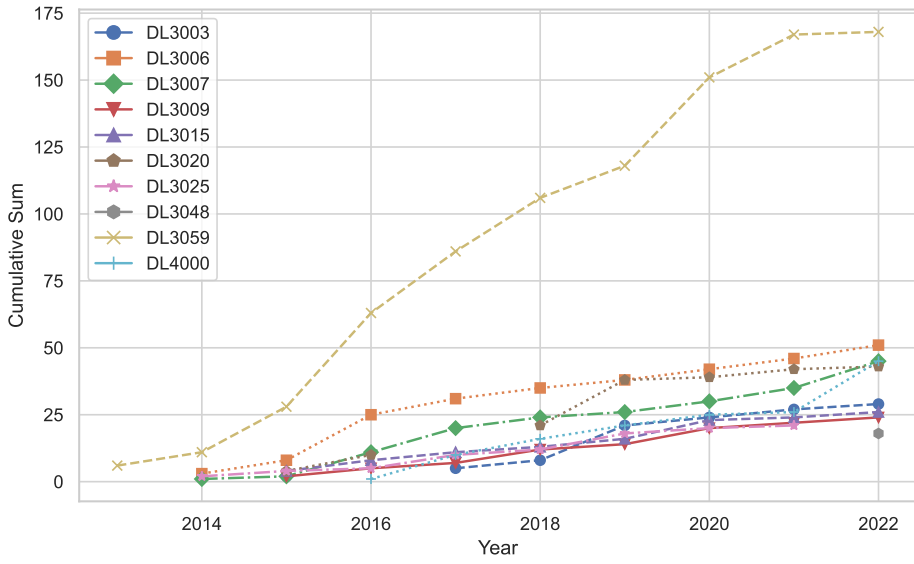


Figure 4.7: Fixing trend over time for the 10 most fixed Dockerfile smells.

over time for the 10 most fixed Dockerfile smells. Also, in this case, we consider only the changes which we manually validated as smell-fixing commits. However, this time, we consider each smell fixed separately. This means that, if a commit fixes 5 smells, we count the commit as 5 different fixes, one for each smell. The most fixed smell is DL3059 – multiple consecutive `RUN` instructions. It is worth noting that we found this fix ∼3 times more frequently than any other fix. This is because we found that, when there are many consecutive `RUN` instructions, developers tend to fix all of the occurrences of this issue in a single commit. Other common fixes are version pinning for base images (DL3006 and DL3007), along with DL4000 – deprecated `MAINTAINER` and DL3020 – prefer `COPY` over `ADD` for files and folders.

We report in Fig. 4.9 the results of our survivability analysis of the smells by plotting the number of fixed smells in different amounts of time (the time is on a logarithmic scale). It is clear that most of the fixes have been performed within 1 day (203 instances). This means that when developers introduce Dockerfile smells, they immediately perform maintenance during the first adoptions. On

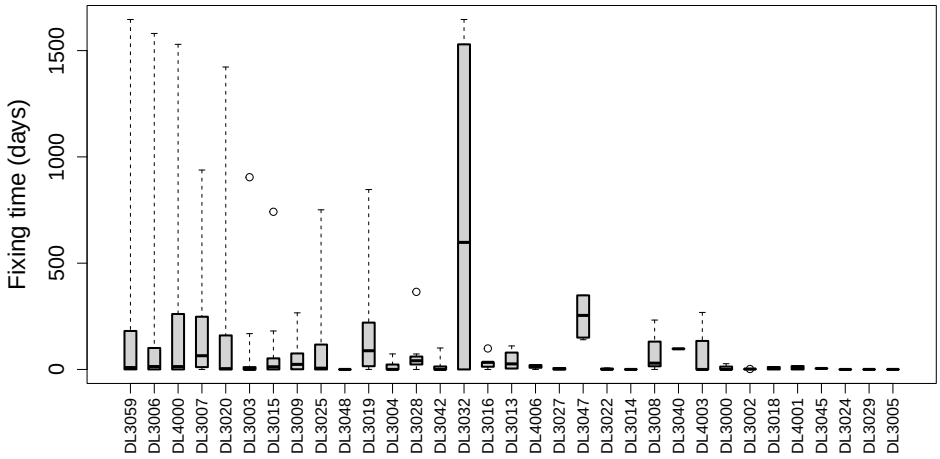


Figure 4.8: Overall fixing time delta (days) among all Dockerfile smells.

the other hand, if a smell survives the first day, it is less likely that it gets fixed later. In fact, according to Table 4.2, the smells that survive the less are DL3048 (incorrect `LABEL` format) and DL3042 (`-no-cache-dir` for `pip install`), which have been fixed in less than one day in most of the cases (100% and 60%, respectively). It is interesting to notice that two similar smells, *i.e.,* DL3006 and DL3007, have largely different survivability. When the `latest` tag is explicitly used (DL3007) instead of being inferred (DL3006), the smell survives ∼5 times more (both in terms of days and commits, as reported in Table 4.2). However, it is worth noting that the effects of both tags are exactly the same.

We evaluated how many smell-fixing commits can be considered *informed*. We consider an informed fix when the developer explicitly mentions that the aim of the fix is to remove bad patterns in the commit message. We found that only 18 out of 336 manually validated fixes are *informed*. The most common smell explicitly addressed by developers is DL4000 (fixed in 4 cases) – deprecated `MAINTAINER`. An example can be found in commit `811582f`, from the repository `webbertakken/K8sSymfonyReact`.[9] Among the remaining ones, DL3025 – JSON notation for `CMD` and `ENTRYPOINT`– (4 cases) and DL3020 – prefer `COPY` over `ADD` for files and folders– (3 cases) are the smells of which developers are more aware.

[9] `https://github.com/webbertakken/K8sSymfonyReact/commit/811582f`

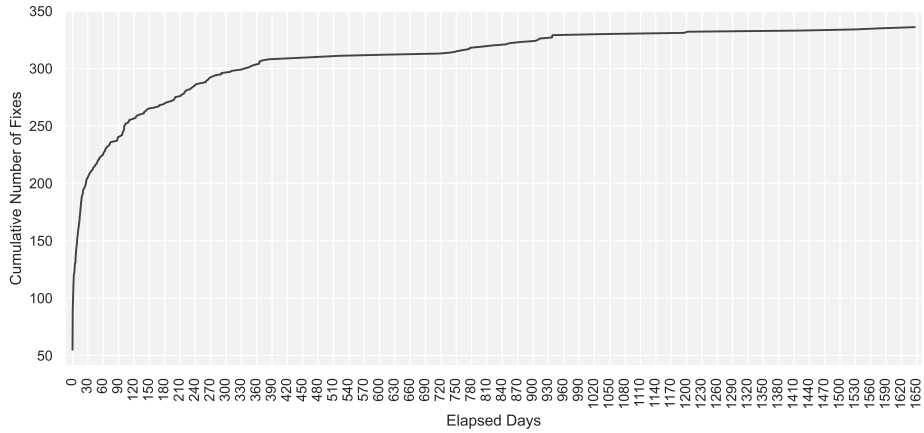


Figure 4.9: Cumulative fixes over time interval (days) among all Dockerfile smells.

As for the *non-informed* cases, mainly developers report that the fix is aimed at (generically) improving the performance of the Dockerfile. Examples are the fixes for rule DL3059 explicitly performed to reduce the Docker image size[10] and the number of layers.[11] In some cases, we found that developers use linters to detect bad practices. Among those, only one commit explicitly mentioned *hadolint*,[12] while in other cases they mentioned the tool *DevOps-Bash-tools*.[13]

In the end, we can conclude that developers have a limited knowledge about Dockerfile best practices, in terms of the quality of the Dockerfile code. This is because they are more interested in the optimization of other non-functional aspects such as build time and size of the Docker image.

[10] https://github.com/KDE/kaffeine/commit/d03145b
[11] https://github.com/Eadom/ctf_xinetd/commit/21f2785
[12] https://github.com/flyway/flyway-docker/commit/3eeabe5
[13] https://github.com/HariSekhon/Dockerfiles/commit/eeab92a

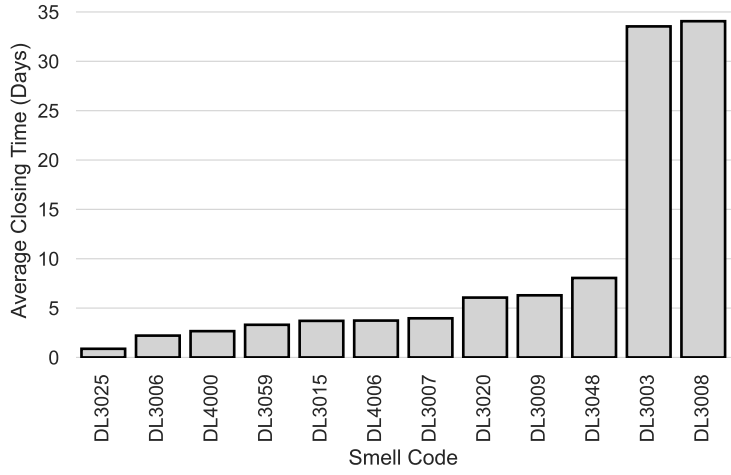
Table 4.2: Summary of fixed Dockerfile smells, reporting the number of fixes (manually validated), median time to fix (in days), and the magnitude of changes performed in the repository until the smell has been fixed (median number of commits). Only smells with at least 5 manually validated fixes are reported.

Rule	Description	# Solved	Days (Med.)	Changes (Med.)
DL3059	Consider consolidation for multiple consecutive `RUN` instructions	168	8.9	4.0
DL3006	Missing version pinning for base image	53	13.7	8.0
DL3007	Avoid to use the latest to tag the version of an image	45	64.6	43.0
DL4000	`MAINTAINER` is deprecated	45	13.5	1.0
DL3020	Use `COPY` instead of `ADD` for files and folders	43	3.8	5.0
DL3003	Use `WORKDIR` to switch to a directory	29	0.2	0.0
DL3015	Avoid additional packages by specifying `-no-install-recommends`	26	12.6	4.0
DL3009	Delete the `apt-get` lists after installing packages	25	23.9	8.0
DL3025	Use arguments JSON notation for `CMD` and `ENTRYPOINT`	21	6.0	2.0
DL3048	Invalid Label Key	18	0.1	0.0
DL3019	Use the `-no-cache` switch when installing packages using `apk`	15	88.0	4.0
DL3004	Do not use `sudo` as it leads to unpredictable behavior	11	0.3	2.0
DL3028	Pin versions in `gem install`	8	41.0	57.0
DL3042	Avoid cache directory with `pip install -no-cache-dir <package>`	7	0.0	0.0
DL3032	`yum clean all` missing after `yum` command	6	597.8	10.0
DL3016	Pin versions in `npm`	5	33.6	4.0

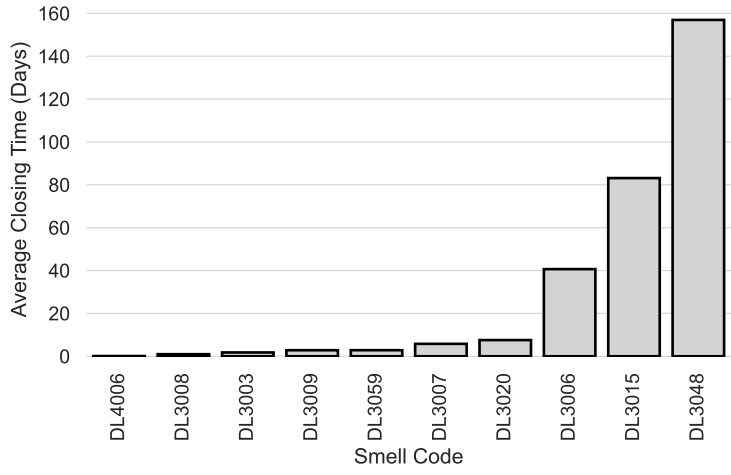
Summary of RQ_1: The most fixed smells are those related to consecutive `RUN` instructions (DL3059), version pinning for the base image (DL3006/DL3007), use of the deprecated `MAINTAINER` instruction (DL4000) along with the usage of `WORKDIR` to change directory (DL3020). The 33.6% of the evaluated commits (1000) actually fixed the smell. Also, most of the smells are fixed immediately after their introduction (within 1 day) and, when this does not happen, they might remain in the repository for a long time (more than 3 years).

4.3.2 RQ_2: *Which Dockerfile smells are developers willing to address?*

In Table 4.3 we report the results of the evaluation performed via GitHub pull requests. In total, we submitted 143 pull requests. The majority of them have been accepted or merged by developers (58%). On the other hand, 23% them have been ignored, while 19% received an explicit rejection from the developers. The smells receiving the highest acceptance rate are DL4000 – deprecated



(a)



(b)

Figure 4.10: Average resolution time (days) for merged pull requests (a) and rejected pull requests (b).

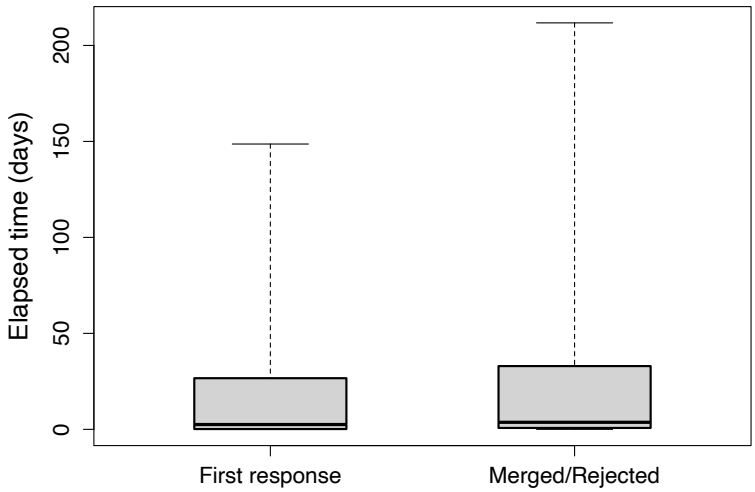


Figure 4.11: Adjusted boxplot of the number of days required for a pull request to obtain a response (left) and to be merged/rejected (right).

Table 4.3: Opened pull requests and their resulting status sorted by number of accepted and merged PRs. The column *Merged** reports the cumulative number of accepted patches (sum of accepted and merged).

Rule	**Ignored**	**Rejected**	**Pending**	**Accepted**	**Merged***	**Assigned**
DL4000	1 (8%)	0	0	0	12 (92%)	13
DL3020	2 (14%)	2 (14%)	0	0	10 (71%)	14
DL3006	2 (23%)	2 (8%)	0	2 (21%)	9 (69%)	13
DL3007	2 (14%)	3 (21%)	0	0	9 (64%)	14
DL3015	3 (23%)	2 (15%)	0	0	8 (62%)	13
DL3025	4 (33%)	0	0	1 (8%)	8 (67%)	12
DL3059	2 (15%)	3 (23%)	0	0	8 (62%)	13
DL3048	2 (22%)	1 (11%)	0	0	6 (67%)	9
DL3009	5 (42%)	3 (25%)	0	2 (17%)	4 (33%)	12
DL3003	1 (14%)	3 (43%)	0	0	3 (43%)	7
DL3008	5 (33%)	7 (47%)	0	0	3 (20%)	15
DL4006	4 (50%)	1 (13%)	0	0	3 (38%)	8
Total	33 (23%)	27 (19%)	0	5 (3%)	83 (58%)	143

`MAINTAINER`– (92%) and DL3020 – prefer `COPY` over `ADD` for files and folders– (71%), followed by rule DL3006 – version pinning for the base image– (69%). This is similar to what we reported for RQ_1, where they resulted to be the most fixed smell among the manually validated smell-fixing commits. This means that developers care about those smells as they frequently fixed them and they are also willing to accept fixes. The smell DL3008 – missing version pinning for `apt-get`– has been the most rejected fix (47% acceptance), with only 3 accepted pull requests, along with smell DL4006 – use of `pipefail` for piped operations– which has been the most ignored one (50%). The low acceptance rate (33%) resulting for smell DL3009 (deletion of `apt-get` sources lists) is surprising, since developers are prone to reduce the image size, as we noticed in RQ_1. Despite this, we can conclude that they do not prefer to remove `apt-get` source lists to achieve this goal.

In Fig. 4.11 we report the adjusted boxplot for the time required for pull requests to get the first response and to be resolved. Additionally, Fig. 4.10 reports the median resolution time, measured in days, of the submitted pull requests by smell type. For both of those figures, we only consider merged and rejected PRs, because they are the ones for which we have a definitive response from the developers. The smell DL3025 – JSON notation for `CMD` and `ENTRYPOINT`– is the one that has been accepted in the shortest time interval, followed by DL3006 – version pinning for the base image– and DL4000 – deprecated `MAINTAINER`. Despite the fixes for DL3020 – prefer `COPY` over `ADD` for files and folders– are the second most-accepted ones, they have a median of 5 days to get accepted and merged. On the other hand, the fixes for DL4006 – use of `pipefail` for piped operations– have been rejected almost immediately by developers. This also happens for DL3008 – missing version pinning for `apt-get`. Finally, we report in Table 4.4 the reasons why developers rejected our pull requests. We assigned one or more categories, for each rejected change, by analyzing the responses for the 27 rejected pull requests. Most of the time, the fix has been considered invalid (22% of cases). This means that the proposed change was not a valid improvement for the Dockerfile. In 11% of cases, the developers did not accept the change as they use the Dockerfile in testing or development environments. The rejections of the fixes for DL3008 are interesting: In 19% of the cases, the changes have

Table 4.4: Categories of reasons why developers rejected our pull requests.

Reason	Involved Smells	Occurrences
Invalid fix	DL3003,DL3007,DL3020,DL3059,DL4006	6
No reason	DL3006,DL3008,DL3015,DL3059	4
Fix not required	DL3008,DL3059	5
Not trusted	DL3007,DL3008,DL3020	3
Testing environment	DL3006,DL3007,DL3008	3
Reduces security	DL3008	2
Development environment	DL3009	2
Vendored dependency	DL3003	1
Potential breaking change	DL3008	1
Unused file	DL3009	1

been rejected because they are not perceived as a concrete fix. Furthermore, the fixes for that smell have been rejected because they could negatively impact the security of the image (8% of cases) or cause a build failure in the future (4% of cases).

Summary of RQ_2: Developers accepted most of the Dockerfile smell fixes we provided (58%) and rejected only a few of them (19%). They particularly liked the fixes for DL4000 (deprecated `MAINTAINER`), DL3020 (prefer `COPY` over `ADD` for files and folders), and DL3006 (version pinning for the base image). Instead, they frequently rejected DL3008 (version pinning for `apt-get` packages) (47%). The reason is that it is seen as a bad practice as it could lead to failures or security issues in the future.

4.4 Discussion

Despite the majority of the submitted pull requests got accepted, there are some specific smells that developers are not willing to address. Looking at Table 4.4, in 5 cases, the fix was rejected because the container was used in a testing or development environment. An example is the fix proposed for DL3009,[14]

[14] `https://github.com/Shopify/semian/pull/484`

where, even if the change can reduce the image size, it negatively impacts the image build time. Thus, for that reason, the change has been rejected. Probably, the concern about build time comes from frequent builds performed for that specific Dockerfile. A different example is the pull request submitted to `envoyproxy/ratelimit`,[15] the reason for the rejection is that developers do not care about the version pinning (DL3007) as they use that Dockerfile for testing and they need to test the latest version of the software. This is not the same for DL3006 when the tag is missing. In that case, developers are more likely to accept the version pinning for the base image (see RQ_1 and RQ_2).

Lesson 1. Developers tend to use the "latest" tag for the base images (DL3007) in order to obtain the latest version of the image, while they are willing to accept the version pinning when the tag is missing (DL3006). However, as the "latest" tag is not immutable, this practice can lead to unexpected behaviors when the base image is updated.

DL3008 constitutes a peculiar case. Fixing such a smell requires developers to pin the version of the `apt-get` packages to make the build more reproducible. Developers, however, believe that doing so might be misleading,[16] or it might make the build more fragile.[17,18] Indeed, this happened for an accepted pull request, where after a month the version pinning for the package `ca-certificates` caused a build failure because the pinned version was not available anymore.[19] Moreover, the smell DL3008 led to interesting discussions. For example, a suggestion was to provide an automated script to periodically pin the package versions when there is an update.[17] For 3 of the proposed fixes, the developers additionally highlighted that they do not trust the change because it has been generated by an automated tool. This happened even if we specified that we manually checked the correctness of the change.

[15] `https://github.com/envoyproxy/ratelimit/pull/411`
[16] `https://github.com/James-Yu/LaTeX-Workshop/pull/3837`
[17] `https://github.com/Lookyloo/lookyloo/pull/663`
[18] `https://github.com/Yelp/aactivator/pull/47`
[19] `https://github.com/FDio/govpp/pull/123`

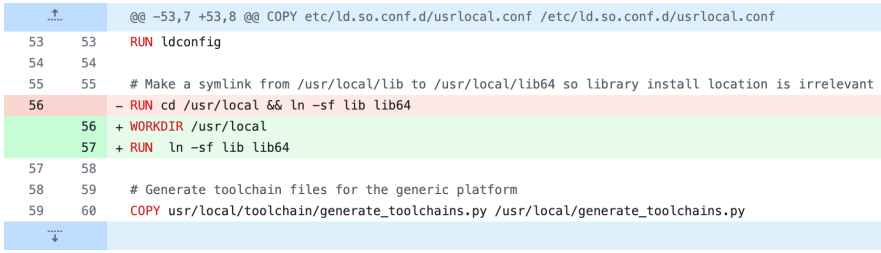


Figure 4.12: Example of a wrong fix for DL3003. In that case, the change of working directory is temporary, and the fix has been rejected.

Lesson 2. Version pinning for OS packages is not considered a good practice. Developers tend to avoid it because (i) they consider it a misleading practice, (ii) it could lead to building failures due to the unavailability of the pinned version, and (iii) missed security updates when the pinned version gets older.

In 6 cases, instead, developers did not perceive the change as correct or sufficient for a fix. This happens, for example, in commits `5531f2e`[20] (DL3020) and `320ba87`[21] (DL4006). An interesting discussion arose for the rejected fix of DL3003.[22] The fix for that smell provides the replacement of `"cd <path>"` with `"WORKDIR <path>"`. However, for that particular case, fixing the smell required putting a `WORKDIR` instruction before the smelly code block and another after to switch back to the previous working directory. This is because the target smelly code temporarily changes the working directory to operate on specific files. In other words, there are cases in which developers believe it is legitimate to change the working directory through `cd` (mostly, when this change is temporary). We report an example in Fig. 4.12, where the fix has been rejected because the change of the working directory is temporary.

We conclude that, in similar cases, the detected smell is a false positive. This is because the fix will increase the number of layers, in addition to redundant instructions. This negatively impacts the code quality of the Dockerfile.

[20] https://github.com/ROCmSoftwarePlatform/Tensile/pull/1707
[21] https://github.com/bupy7/xml-constructor/pull/6
[22] https://github.com/NUbots/NUbots/pull/1063

Comparing the results from RQ_1 and RQ_2, we can conclude that there are no big differences between the fixes that developers have applied and the changes that we propose via pull requests. The most performed fixes, which are also in the most accepted pull requests, are those related to deprecated `MAINTAINER` (DL4000), version pinning for the base image (DL3006), and multiple consecutive `RUN` instructions (DL3059). There is a difference in terms of the most fixed one. While in the wild developers tend to fix more DL3059, in our pull request the most fixed one is DL4000. As also shown in RQ_1, they pay more attention to performance improvements over code quality, for which they are not fully aware of what is the current writing best practices.[23] In fact, DL4000 is purely related to writing best practices and does not affect performance. When faced with a ready-to-use fix, however, they tend to prefer the ones that more likely will not disrupt the Dockerfile. In general, developers keep more attention to the impact of the change on the build process and the image size, instead of the impact on the quality of the Dockerfile code. Reporting an example among the accepted pull requests, we have the fix proposed for the smell DL3015 (`-no-install-recommend` flag for apt),[24] where the developers explicitly asked to fix another Dockerfile affected by the same smell because it decreases the size of the built image.

Lesson 3. Developers are not fully aware of the best practices for writing Dockerfiles, and they tend to prefer performance improvements over code quality.

Additionally, it is interesting to analyze more in depth the differences in terms of performed fixes for DL3048 (incorrect `LABEL` format) and DL4000 (`MAINTAINER` is deprecated, replace with `LABEL`). Actually, there are two possible ways to format Dockerfile labels. The first one follows the standard format defined by *opencontainers*,[25] which is also suggested for DL4000 in the official Docker documentation [21]. The second is more general and does not enforce a pre-defined format. It is reported in the *hadolint* documentation,[26] which also is reported in the official

[23] https://github.com/riga/law/pull/152
[24] https://github.com/lablup/backend.ai/pull/1216
[25] https://specs.opencontainers.org/image-spec/annotations/
[26] https://github.com/hadolint/hadolint/wiki/DL3048

Docker documentation as examples of LABEL instructions.[27] The fixes that we analyzed in RQ_1 that follow the first format are limited only to one repository.[28] In other cases, developers adopted the second format.[29] The fixes proposed via pull requests, instead, follow the second format where for DL4000 we got the highest acceptance rate. This is probably because the second format is more general, avoiding unnecessary constraints and changes on the LABEL instructions.[30]

Moreover, while in this context the fix is still sufficient to correct the smell, in other contexts our fixing procedure could not be correct. The most evident case is for the smell DL3059 (multiple consecutive RUN instructions). In fact, open-source developers tend to fix it mainly by compacting the installation of software packages.[31] In our pull requests, instead, we merge all the subsequent RUN instructions until a comment or a different instruction is found. This could mean that a more complex and informed fixing procedure should be adopted in order to better improve the size and performance of Dockerfiles. Thus, a more advanced approach in that direction could be useful to improve the fixing procedure, taking also into account the aspects that developers are interested to improve (image size and build time).

To this aim, considering the scenario in which we are using a `debian` base image, an advanced approach to fix smell DL3059 could be a heuristic that (i) selects all the RUN instructions that are aimed at installing dependencies, (ii) extracts the list of such dependencies, taking also into account if they require external sources lists, and (iii) combine all those installations into a single RUN instruction at the top of the Dockerfile. In this way, the re-build time will be reduced thanks to the layers caching system. At the same time, the image size will be reduced since there will be fewer layers and less space wastage (*e.g.,* package cache). For smell DL3003, instead, an advanced fixing approach should target the *bash* code to correct the usage of the pattern "RUN `cd ...`", rather than using WORKDIR.

[27] https://docs.docker.com/engine/reference/builder/#label
[28] https://github.com/HariSekhon/Dockerfiles/commit/f329b94
[29] https://github.com/scossu/lakesuperior/commit/a552ff7
[30] https://github.com/hpc/charliecloud/pull/1628
[31] https://github.com/hpc/charliecloud/commit/aae89d7

In the example reported in Fig. 4.12, the smell could be fixed by using the absolute paths instead of the relative paths for the command (*e.g.,* `"RUN ln -sf /usr/local/lib /usr/local/lib64"`). While this can be done in this case, there are other scenarios in which this could be detrimental. For example, if a custom script writes the output files in the current directory, it is still necessary to use `cd` before running it. Thus, such a fixing procedure should be applied only for specific bash instructions patterns (like the previously-mentioned one).

Lesson 4. A more advanced fixing procedure is required for some types of smells (*e.g.,* DL3003 – Use `WORKDIR` to switch to a directory– and DL3059 – multiple consecutive `RUN` instructions), *i.e.,* taking into account the context in which the smell is found.

4.5 Threats to Validity

Construct Validity. The threats to construct validity are about the non-measurable variables of our study. More specifically, our study is heavily based on the rule violations detected by *hadolint*. Other tools are able to detect bad practices in Dockerfiles, such as *dockle*.[32] We choose *hadolint* which is commonly used in the literature [13, 129, 66, 27] and also in enterprise tools for code quality.[33] However, *hadolint* could lead to false positives or can miss some smells.[34] The manual evaluation we performed on the smell-fixing commits validated the identified smells and those that have been removed. During that evaluation, we noticed that *hadolint* mainly fails to detect the rule DL3059 (consecutive `RUN` instructions). To reduce this impact of this threat on our study, we manually annotated the lines in which the smell was present.

Internal Validity. The threats to internal validity are about the design choices that we made which could affect the results of the study. In detail, we used as a study context a sample of repositories extracted from the dataset provided by Eng *et al.* [27] by considering only those having stargazers count greater or equal to 10. This is commonly used in the literature to avoid toy projects [17].

[32] `https://github.com/goodwithtech/dockle`
[33] `https://github.com/codacy/codacy-hadolint`
[34] `https://github.com/hadolint/hadolint/issues/693`

There can be a bias in the selected smells for our fix recommendations. We selected the most occurring smell as described in the analysis of Eng *et al.* [27]. We assume that an automated approach would have the biggest impact on the smells that occur more frequently. Also, at least for some of them, the reason behind the fact that they do not get fixed might be that they are not trivial (*i.e.,* an automated tool would be helpful). The fixing procedure for some of the selected smells can be wrong, and some smells might not get fixed. We based the rules on the fixing procedure on the Docker best practices and on the *hadolint* documentation. Still, to minimize the risk of this, we double-checked the modifications before submitting the pull requests and manually excluded the ones that make the build of the Dockerfile fail. Thus, we ensured the correctness of the fixes generated by DOCKLEANER, submitted via the pull requests, for the cases evaluated in our study. However, it is still possible that the tool produces wrong fixes for other Dockerfiles. For example, the version pinning fixes could fail in the cases in which the package is not reachable (*i.e.,* DL3008), or the Docker image digest is not available in DockerHub (*i.e.,* for smells DL3006 and DL3007). It is worth noting, indeed, that our aim is not to evaluate the tool, but rather to understand if developers are willing to accept fixes. Moreover, there is a possible subjectiveness introduced of the manual validation of the smell-fixing commits, which has been mitigated with the involvement of two of the authors and the discussion of the conflicts. Also, it is important to say that the two evaluators have more than 3 years of experience with Dockerfiles development and Docker technology in general, allowing them to have a good understanding of the smells and the applied fixes. Finally, we performed the selection of the *last-smell-introducing* commits by using the *git blame* command on the smelly lines identified by *hadolint*. Since *hadolint* can fail to detect some smells, in some cases, the lines impacted by the fix are different from the ones identified by *hadolint*. This means that we got some false positives while we identify the *last-smell-introducing* commits. Since our results showed that Dockerfiles are not frequently changed, we believe that the impact of this threat is limited.

External Validity. External validity threats concern the generalizability of our results. In our study, we considered a sample of repositories from GitHub containing only open-source Dockerfiles. This means that our findings might not

be generalized to other contexts (*e.g.,* industrial projects) as developers could handle smell differently.

4.6 Final Remarks

Best practice violations, namely Dockerfile smells, are widely spread in Dockerfiles [13, 129, 66, 27]. In this chapter, we proposed an empirical study to evaluate the Dockerfile smell survivability by analyzing the most fixed smells in open-source projects. We found that Dockerfile smells are widely diffused, but developers are becoming more aware of them. Specifically, for those that result in a performance improvement. In addition, we evaluated to what extent developers are willing to accept fixes for the most common smells, automatically generated by a rule-based tool. We found that developers are willing to accept the fixes for the most commonly occurring smells, but they are less likely to accept the fixes for smells related to the version pinning of OS packages. To the best of our knowledge, this is the first in-depth analysis focused on the fixing of Dockerfile smells.

We reported some lessons learned on top of the obtained results. In summary:

- Smell DL3007, *i.e.,* use specific version pinning for the base image, could be misleading in some cases. Developers tend to use the "latest" tag with the intention to take the most up-to-date version of the base image. However, using that tag does not ensure reproducibility and could lead to unexpected behaviors when the base image changes.;
- Version pinning for OS packages (DL3008) is not considered a good practice, since it could lead to build failures or security issues in the future (if not frequently updated);
- As a confirmation of the findings reported in Chapter 3, developers prefer performance improvements over code quality. They tend to prefer fixes that improve that aspect;
- Another complementary finding of the results of the previous chapter, is that some smells (*e.g.,* DL3003 – use `WORKDIR` – and DL3059 – merge consec-

utive RUNs –) require a more advanced fixing procedure, taking into account the context in which they happen.

Nevertheless, we believe that there is still room for future improvements in this direction. In particular:

- An important contribution to the topic of Dockerfile smells is the empirical validation of the effectiveness of *hadolint*. This will help not only to improve the tool but also to better understand the impact of false positives and false negatives on the results of the studies in this domain;
- DOCKLEANER only provides the fix for a limited number of smells (12). A possible future work could be to extend the tool with more rules included in the *hadolint* catalog (66 in total). Additionally, the fixing procedure could be improved by taking into account the context in which the smell is found;
- The same can be done with the existing tools and approaches able to detect smells: Considering the context in which the smell is found might help to reduce false positives.

Acknowledgments

We thank the student Alessandro Giagnorio (University of Molise, Italy) for implementing a preliminary version of DOCKLEANER.

CHAPTER 5

Improving the Build Time and Size of Docker Images: Strategies from Developers

Having small Docker images, in terms of storage size, is desirable because it allows using less resources on the deployment server and, at the same time, reduces the deployment costs. At the same time, reducing the time needed to build an image from the source Dockerfile is important in a scenario in which developers frequently deploy their product (*e.g.,* in a DevOps environment). The results obtained in the previous chapters showed that developers pay more attention to the functional aspects of Dockerfiles, while they might not be entirely aligned with the best practices suggested by Docker and the literature. In fact, they prioritize performance and security (Chapter 3), and tend to apply changes aimed at improving those aspects (Chapter 4).

As we previously discussed, different works in the literature investigated the performance-related aspects of Docker images. For example, Zhang *et al.* [140] reports that slow build time in CI/CD pipelines (which includes the build of Dockerfiles) leads to poorer developers' work efficiency. Ksontini *et al.* [58] found that ∼19.7% of the Dockerfile refactoring operations performed by developers

are aimed at improving such aspects. While this study provides valuable insights in the practices used by developers to improve both the build time and the size of Docker images, its goal was more generic (*i.e.,* it was aimed at studying all the refactoring operations performed by developers). Thus, the authors ended up analyzing only 38 commits related to performance improvement. Besides, we do not know what impact the refactoring operations made by developers have. We hypothesize that such a previous work only scratched the surface of what developers do to address performance issues in Docker images. Despite there are several tools and approaches aimed at optimizing Docker images, this implies that they need to be executed every time a new build is completed. Ideally, the source Dockerfile should be reasonably optimized already to avoid such an overhead.

To fill this gap, in this chapter, we present an extensive empirical study in which we aim (i) to understand what developers do to improve the performance of Docker images and (ii) what impact such operations have, in practice. Specifically, we consider the build time and the image storage size as the performance-related aspects. The context of the study is composed of instances from the dataset proposed by Eng *et al.* [27], which contains commits aimed at modifying Dockerfiles in GitHub. We ran two queries, on that dataset, to extract changes aimed at improving either the build time or the image size. We manually analyzed a significant sample of such commits (∼1,200), to manually tag them with a high-level description of the operations made by developers. As a result, we selected a total of 383 commits (528 tags). We defined a taxonomy of 54 refactoring operations made by Dockerfile developers to improve build time and image size. We found that most of the changes are aimed at de-bloating the image (45% of the commits analyzed) by removing unnecessary files and, thus, reducing the image size. Besides, developers tend to modify the Dockerfile architecture (32% of the changes) and to foster the use of caching mechanisms (18% of the changes) to reduce the build time. We also tried to go further and understand what impact such modifications have in practice. We built the Dockerfiles we considered in our study both before and after the change that was supposed to improve the build time or the image size and measured the improvement. Unfortunately, we found that the large majority of the Dockerfile snapshots we considered (75%) could not be automatically built (*e.g.,* because of missing packages or because

some environment variables needed to be manually set). As for the ones we could build, we observed that changing the base image with a more adequate one has the biggest impact both on image size and build time. Similarly, removing unnecessary files and reducing the number of layers benefits both performance-related aspects. However, removing the cache of package managers generally has a limited impact on the final image size.

Our results might be useful to both developers and researchers. We provide developers with a catalog of operations that they can apply to reduce the image size and build time of Docker images. On the other hand, we provide researchers with useful insights that they can use to devise new approaches aimed at refactoring Docker images for performance improvement (*e.g.,* by automatically suggesting optimal base images).

The rest of the chapter is organized as follows. Section 5.1 presents the methodology and details of our empirical study, including the data collection process, experimental design, and techniques applied. In Section 5.3 we discuss the results, followed by threats to validity in Section 5.4. Finally, in Section 5.5 we summarize the final remarks along with future research directions.

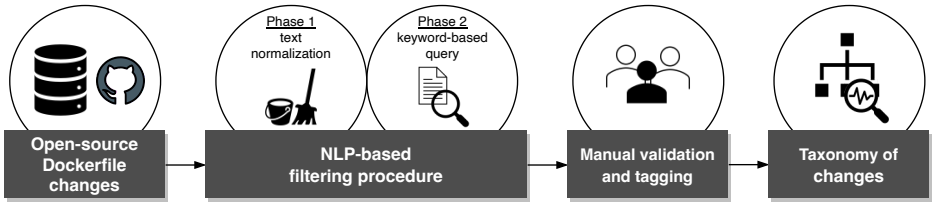


Figure 5.1: A summary of the experimental procedure applied to extract the performance-related changes analyzed in our study.

5.1 Empirical Study Design

The *goal* of our study is to understand how Docker developers change Dockerfiles to improve the build time and size of the resulting Docker images and how such changes impact those quality aspects. The *perspective* is of both researchers

and developers interested in improving those aspects when writing Dockerfiles. The *context* consists of 383 commits coming from 369 open-source repositories.

In detail, the study addresses the following research questions:

RQ_1: *Which changes do developers apply to improve the image size and build time of Docker images?* We want to investigate the developers' activity on existing Dockerfiles to capture the common change patterns applied to improve the image size and build time of the resulting images;

RQ_2: *To what extent does the applied changes impact image size and build time of Docker images?* We want to conduct a preliminary analysis, *i.e.,* limited to the changes identified in the previous RQ, to measure their impact on the final Docker image. This is to quantify the magnitude of improvement when applying those changes.

5.1.1 Data Collection

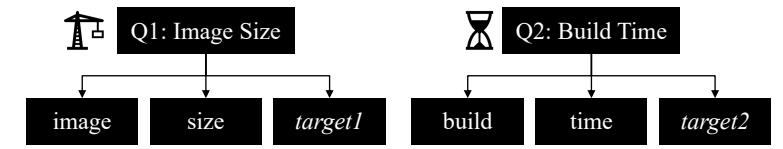
The context of our study is represented by commits extracted from the dataset proposed by Eng *et al.* [27]. It is currently the largest Dockerfile dataset present in the literature as compared to others [66, 43]. It contains the change history of Dockerfiles extracted from all the open-source GitHub repositories up to 2021. The data extracted regard a total of 1.9M repositories, for a total of 11.5M commits related to about 9.4M Dockerfiles. For our study, we selected a subset of those changes, *i.e.,* the ones aimed at improving the build time and image size of the resulting Docker images. To do this, we rely on the commit message, selecting those in which developers explicitly report the intention of reducing the build time or the size.

We defined a heuristic approach to filter commits based on what developers reported in the commit message using Natural Language Processing (NLP) techniques. In particular, we defined an NLP pipeline using *Node.js* with the libraries *Natural*[1] and *Snowball*.[2] The pipeline is composed of two phases: a text pre-processing phase, followed by a keyword-based query to select only the relevant commits. The entire process is reported in Fig. 5.1.

[1] `https://www.npmjs.com/package/natural`
[2] `https://www.npmjs.com/package/snowball`

Text Pre-processing. The first phase includes the usage of different NLP techniques to prepare and normalize the input commit messages. First, to apply NLP processing techniques it is necessary to split the plain text into single tokens (*i.e.,* words). We achieve this by applying *word tokenization* to split the input commit message in tokens (*i.e.,* single words). Follows a *stop-word* filtering, in which the tokens that are not informative are removed. As a last step, we apply a stemming procedure that reduces each word to the root form by removing suffixes and prefixes. This allows the normalization of all the tokens and simplifies the keyword-based selection phase.

Keyword-based Selection. To filter only the performance-related commits improving build time and reducing Docker image size, we applied a keyword-based filter to the processed commit messages using two different queries, one for each scope. The queries have been defined via a manual process in which (i) we pick one or more keywords, (ii) perform the query, and (iii) manually verify the resulting commits to adjust the query (*i.e.,* to avoid false positives and add new keywords). This process was executed for each one of the queries until there were no other more keywords or modifications to add. In the end, we defined the two queries reported in Fig. 5.2.




```
target1 = [ decrease,reduce,cut,small,drop,optimize ]
target2 = [ speed up,decrease,reduce,cut,small,drop,diminution ]
```

Figure 5.2: The keyword-based filter used to filter performance commits.

When applying query *Q1*, a commit is selected if it contains both the tokens *image* and *size*, and one of the tokens from the set *target1* (*e.g.,* decrease). Likewise, when we apply query *Q2*, a commit is selected if it contains both the tokens *build* and *time*, plus one of the tokens from the set *target2* (*e.g.,* drop).

5.1.2 Experimental Procedure

To answer RQ_1, we applied our filtering procedure to all the commits contained in the dataset by Eng *et al.* [27]. Thus, we obtained a subset of 1,200 commits matching at least one of the two queries reported in Fig. 5.2. The process is performed by randomly picking one commit at a time, for each query, until a sample of 400 validated commits is reached (5% margin of error, 200 per query) in line with previous work [58]. The validation was performed manually by two of the authors, from now *evaluators*, on the commit messages and the change. During the process, the changes that are (i) false positives wrongly selected by our heuristic approach, (ii) not publicly available anymore, and (iii) duplicated (*i.e.,* from forked repositories). An example of a discarded false positive is commit `0313392` from the repository `hypothesis/bouncer`,[3] where the author explicitly reported in the commit message: "[...] the resulting image is slightly larger [...] but there is plenty of room to reduce the image size in other ways". The query matched the words *reduce*, *image*, and *size*, but the commit has been excluded since it negatively impacts the image size. Another example is commit `bb96380`,[4] which aims at reducing the image size according to the commit message. However, the change itself has no impact on the image size, since it only adds the `yum update` operation in an existing `RUN` instruction. In the end, the valid commit we considered is 383, corresponding to 369 repositories. In addition, during the manual validation, the evaluators explicitly annotated if the commits only reduce the build time, decrease the image size, or both: Indeed, some of the commits we selected for one of the aspects were also aimed at improving the other one.

Next, the two evaluators manually annotated the commits with one or more tags to describe the type of change. They obtained a "substantial" agreement rate ($\kappa = 0.71$) [60]. Followed a cross-validation phase, which involved both the two original evaluators and another additional evaluator (*i.e.,* three, in total) to discuss and resolve the conflicts.

Finally, following a card-sorting inspired approach [113], two evaluators categorize the annotated tags and organize them in a taxonomy. More specifically, the two annotators started with a first-round aimed at abstracting the tags. Then,

[3] `https://github.com/hypothesis/bouncer/commit/0313392`
[4] `https://github.com/cropgeeks/docker/commit/bb96380`

followed two more rounds in which they grouped similar changes. In the end, they discussed the obtained tags and ordered them into macro and subcategories to build a first draft of the taxonomy. A final round followed in which the two annotators double-checked each tag and the assigned category, by renaming and reordering them when needed. After this, the final version of the taxonomy has been obtained. For each change reported in the taxonomy, we report the number of occurrences and which aspect it improves (*i.e.,* build time, size, or both). We report and discuss in detail the taxonomy by reporting some representative examples for each category. The entire experimental procedure is summarized in Fig. 5.1.

To answer RQ_2, we select a subset of the evaluated changes in order to build the Docker images before and after the patch to measure the impact in terms of size and build time. The aim is to conduct a preliminary analysis to measure the impact of the changes we identified in RQ_1 on the final Docker image. In detail, we selected all the annotated commits in RQ_1 corresponding to only one tag (*i.e.,* 226, in total) to ensure that the change has only one specific aim. Next, for each resulting commit, we automatically (i) fetched the whole repository, and (ii) if the commit modifies only one Dockerfile, we checked both the revision in which the improvement has been applied and the one before. To achieve this, first, we cloned the repositories, then we used *PyDriller*[5] to extract the commit metadata. We searched among the modified files for Dockerfiles (*i.e.,* the file name in lowercase matches the pattern "%dockerfile%"). In case of merge commits, we select the commit corresponding to the last modification of the analyzed Dockerfile. In the end, we obtained after and before-change revision pairs, where the change is aimed at improving either the image size or build time. First, we perform a test build for the Dockerfile snapshot after and before the change. This is to (i) verify that the build is working, and (ii) check and pull the base image used in the Dockerfile. Subsequently, we systematically performed the build of the Dockerfiles snapshots after and before the change. We executed those builds on a dedicated Ubuntu server VPS, with 6 vCPUs and 16GB of RAM. We built only one snapshot at a time to avoid any kind of bias in the measurement, and we used a paid DockerHub plan to mitigate the pull rate limit

[5] `https://github.com/ishepard/pydriller`

of the Docker Hub APIs. We used two different procedures based on the type of change. As for changes impacting the build time, we repeated the build 10 times both before and after and we measured the time needed in both cases. We did this because the build time might be slightly different among different builds because of other background processes running. Anyway, to reduce their impact, we ran the experiment on a dedicated machine and made sure that no unnecessary process was executed in the background during the experiment. Since the base image has been pulled during the test build, the build time is not affected by the time required for downloading it, which depends also on the base image size. We perform each build using the option `--no-cache`: This ensures that the build was performed for each layer without using the layer caching system of Docker. We did this to avoid that layers built in the previous run being re-used, thus making the second and successive runs much shorter than the first one. We used a build timeout of 1600 seconds to handle cases in which the build stuck. This is used in studies performing a similar procedure [13, 97]. Instead, for the changes impacting the image size, we executed a single clean build of the Dockerfile snapshots in their original build context (*i.e.,* source repository). In this case, indeed, the measure we take is deterministic. We executed separately the build for the changes improving the image size and the build time, for a total of 123 and 105, respectively, with two changes improving both image size and build time.

We discarded the commits in which the build failed either for the snapshot after or before the change. Investigating the reasons behind the failed build, we noticed that in a lot of cases, the build failure is related to package installation errors (*e.g.,* not found in package repositories). Enabling those builds requires a manual intervention to fix those issues, and thus altering the original Dockerfile. In other cases, the Dockerfile requires specific settings (*e.g.,* build argument or environment variable). Since we are not able to define those settings or, otherwise, they could lead to different outcomes (*i.e.,* they can influence the build behavior-inducing bias), we considered only the snapshots that we are able to build without specific settings.

In the end, for each commit pair, we obtain and report the image size before and after the change, and the build time resulting from the 10 different builds. We

report the average improvement for changes aimed at improving image size and build time. In addition, for changes improving build time, we perform a Mann-Whitney-U test to measure the significance of the change improvement [128]. The null hypothesis is that there is no difference between the build time before and after the change. We also report a quantitative measure for the effect size [128] (*i.e.,* Cliff's Delta[6]).

5.1.3 Data Availability

The tagged data and the raw results of our experiments are publicly available in our replication package [3].

[6] https://pypi.org/project/cliffs-delta/

Figure 5.3: Taxonomy of changes reducing build time and image size for Dockerfiles and Docker images. The total number of occurrences are reported for each category and sub-category. Additionally, each attribute has a badge if the change reduces the image size (**S**), the build time (**B**), or both (**S** and **T**).

5.2 Empirical Study Results

In this section, we report and discuss the results of our study.

5.2.1 RQ_1: *Which changes do developers apply to improve the image size and build time of Docker images?*

We report in Fig. 5.3 the taxonomy of changes applied by developers to reduce the build time and image size of Dockerfiles and Docker images. We report, for each category, the number of occurrences of that type of change. Moreover, we report the single change as an *attribute* having the number of specific occurrences and a badge indicating if the change reduces the image size (**S**), the build time (**T**), or both (**S** and **T**). We assigned a total of 528 tags grouped in 4 different macro-categories, as described in the following.

- *Debloating*: Describes changes aimed to remove or avoid unnecessary files and dependencies during the build process of the Dockerfile;
- *Dockerfile Architecture*: Describes changes aimed at improving Dockerfiles by performing structural modifications, such as joining instructions or changing the base image.
- *Caching*: Describes changes aimed at using the caching procedure during the build of Docker images in a more efficient way;
- *Tweaks*: Describes changes aimed at optimizing the usage of tools for build and dependency installation;

The most frequent changes are categorized as Debloating (45%), followed by Dockerfile Architecture (32%). Categories Caching and Tweaks are the less frequent (18%) and 5%). The changes in Debloating mainly impact the final image size, while those in Tweaks and Caching the build time. On the other hand, changes in Dockerfile Architecture mostly impact both aspects. In the following, we describe them in detail by reporting some examples.

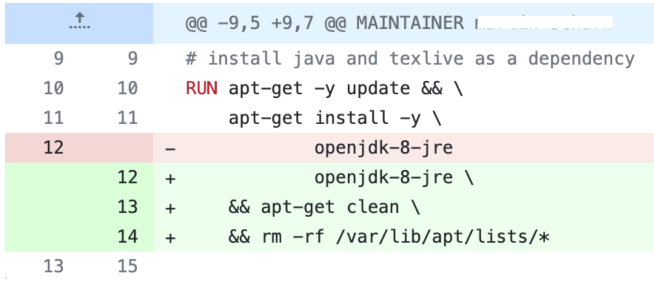


Figure 5.4: Example of a "Debloating" change, aimed at removing the apt cache and sources lists.

Debloating

To reduce the clutter in Docker images, developers remove the additional data used by package managers (86 occurrences), *i.e.,* by performing a cleanup of the packages, data, and cache during the installation of dependencies. This kind of change mainly aims to reduce the image size. For example, when installing a dependency using the `apt` package manager, a common pattern is to run `apt-get clean`, remove `apt` lists and the used cache. In some cases, running additionally the command `apt-get autoremove` could also contribute to remove unnecessary files. In Fig. 5.4 we report an example for *remove* `apt-get` *lists* and `apt-get` *clean/autoclean* changes, proposed in commit `fdcebf4`.[7] Specifically, `apt-get clean` and the removal of the sources lists have been added right after calling `apt-get install` to install packages.

Other types of changes are focused on the removal of wasteful data (80 occurrences). Examples are excluding development dependencies from the final Docker image or removing temporary files (*e.g.,* removing `/tmp/*` and `/var/tmp/*`). Another typical change consists in the introduction of a `.dockerignore` file. Such a file contains patterns of files that should be excluded from the build context. This change has a positive impact on both the build speed (*i.e.,* smaller context to handle) and the size, as it might reduce the number of files that are copied in the image through `COPY` or `ADD` instructions. Also, excluding files from the build context allows to speed up the overall time required for the build.

[7] https://github.com/binfalse/docker-debian-testing-java8/commit/fdcebf4

```
...  ...   @@ -1,10 +1,8 @@
 1        - FROM ubuntu
      1   + FROM dockheas23/ut-haskell-base:v1
 2    2     MAINTAINER <...>
 3        - RUN apt-get update && apt-get install -y cabal-install ghc git libghc-zlib-dev
 4    3     RUN git clone https://github.com/Dockheas23/ut-haskell /opt/ut-haskell
 5    4     RUN mkdir /opt/ut-haskell/log
 6    5     RUN touch /opt/ut-haskell/log/{access,error}.log
 7        - RUN cabal update
 8    6     RUN cd /opt/ut-haskell && cabal install
 9    7     EXPOSE 8080
10    8     CMD cd /opt/ut-haskell && /root/.cabal/bin/ut-haskell -p 8080
```

Figure 5.5: Example of a "Dockerfile Architecture" change in which the base image is replaced with one including `haskell` dependencies.

Finally, developers often aim to reduce the number of layers in the Docker image (71 occurrences) to reduce both the image size and the build time. To do this, in most of the cases, developers join several `RUN` instructions in a single one.

Dockerfile Architecture

The most frequent type of modification from this category is the change of the base image (106 occurrences). Developers usually prefer a smaller variant of the same Docker image (64 occurrences), *e.g.,* `python:3.11-alpine` instead of `python:3.11`. In particular, in 38 of these cases, developers adopted the *alpine* flavor of the same base image, which is typically smaller. Alternatively (42 occurrences), they switch to a completely different base image because it is smaller (*e.g.,* from `ubuntu` to `debian`), or else reduces the build time because it embeds some required dependencies (*e.g.,* from `ubuntu` to one already including `python`). Thus, the installation steps for those dependencies are removed from the Dockerfile reducing the overall build time. An example of a *Base Image* change is reported in Fig. 5.5 (from commit `0879533`[8]). In detail, the generic `ubuntu` base image is replaced with a more specific one containing already the required `haskell` dependencies, avoiding installing them after in the Dockerfile.

Another frequent operation is adding or modifying *Build Stages* (49 occurrences) of the Dockerfile. In this case, developers more often restructure the

[8] `https://github.com/Geeroar/ut-haskell/commit/0879533`

```
...  ...  @@ -1,9 +1,19 @@
1    1    FROM python:latest
2    2
3         - COPY . /
     3    + COPY requirements.txt /
4    4
5    5    RUN pip install -r requirements.txt
6    6
     7    + COPY ./Mongo /Mongo
     8    + COPY ./Postgres /Postgres
     9    + COPY ./Neo4j /Neo4j
     10   + COPY ./Enums /Enums
     11   + COPY ./ElasticSearch /ElasticSearch
     12   + COPY send.py /
     13   + COPY settings.py /
     14   + COPY api.py /
     15   +
     16   +
7    17   EXPOSE 5000
8    18
9    19   CMD [ "python", "./api.py" ]
```

Figure 5.6: Example of a "Caching" change in leveraging the layer caching to optimize the build.

Dockerfile enabling multi-stage builds (46 occurrences). This consists of grouping the Dockerfile instructions in separate stages, corresponding to isolated steps of the build process executed in a new Docker image. This allows to discard temporary files, reducing the size of the final image (used as the final step), and easily handling the build dependencies (*e.g.,* using a pre-built image) reducing also the build time.

Finally, developers radically change the way they containerize their software by splitting a single Dockerfile into several ones. As an example, they sometimes extract several instructions and define a new Dockerfile; the resulting image is used as the base image of the remainder of the Dockerfile, which is the main one. This type of modification allows developers to reduce the build time of the main image (since part of the build is now a Docker image that is cached and rarely requires to be built again) and the build size (since the base image can be further optimized, *e.g.,* by compacting its layers).

Caching

The changes in this category consist mainly of modifying the instruction order (*Sort Instructions*, 81 occurrences) to improve the usage of the layer caching during the build. This means, for example, moving the `COPY` instruction at the bottom of the Dockerfile (43 occurrences). Since the source files are those that usually change, it will result in a faster build, especially during development. Another common operation is to copy and install the requirements before copying sources or performing other operations, in order to reduce the build time (21 occurrences). The most common example (Fig. 5.6) is to copy only the Python `requirements.txt` before installing the requirements separately from the other sources. This will leverage the Docker cache speeding up the next build when updating the source code files.[9]

Interestingly, developers need to avoid caching in package managers as a generic mechanism sometimes to reduce image size (8 occurrences). Indeed, while caching at the Docker level is positive (*e.g.,* the previously-mentioned layer caching mechanism), using the cache of the package managers during the build is mostly negative. Such a mechanism, indeed, increases the image size (the cache is inside the resulting Docker image) but it does not speed up the build since those files will be discarded in the next build (or the layer caching mechanism will take over whatsoever). This is why developers tend to use options to avoid caching in `apk`, `pip`, and `bundle`.

Finally, developers apply more specific changes to enable the use of caching in a few cases (7 occurrences). For example, we found that in some cases (3 occurrences), developers adopt an external `VOLUME` with the dependency cache and mount it during the build to speed it up.

Tweaks

The less occurring changes are those aimed at optimizing the usage of tools (*Tool-specific*, 5 occurrences) and *Install Operations* (21 occurrences). An example of the first is switching to a more efficient tool (*i.e.,* from `npm` to `yarn`). This change helps to reduce the build time[10]. For the latter, developers avoid

[9] `https://github.com/detiuaveiro/social-network-mining/commit/020d5c3`
[10] `https://github.com/pnpm/benchmarks-of-javascript-package-managers`

```
@@ -27,8 +27,8 @@ RUN git clone https://github.com/Y-modify/deepl2 --depth 1 \
27 27       && cd deepl2 \
28 28       && git clone https://github.com/openai/baselines --depth 1 \
29 29       && sed -i -e 's/mujoco,atari,classic_control,robotics/classic_control/g' baselines/setup.py \
30    -     && pipenv install baselines/ \
31    -     && pipenv install
   30 +     && pipenv install baselines/ --keep-outdated \
   31 +     && pipenv install --keep-outdated
32 32
```

Figure 5.7: Example of a "Tweaks" change avoiding the upgrade of the dependencies installed using `pip`.

dependency upgrades (5 occurrences) or they compile sources for fewer versions or targets (6 occurrences) to reduce build time overheads. We report an example for *Avoid Dependency Upgrade* in Fig. 5.7, in which the flag `--keep-outdated` prevents the upgrade of `pip` packages before installing.[11]

Summary of RQ_1: Four main types of changes are performed by developers to improve the image size and build time of images: Those for *Debloating* (45%), changing the *Dockerfile Architecture* (32%), optimizing *Caching* (18%), and installation *Tweaks* (5%).

5.2.2 RQ_2: *To what extent does the applied changes impact image size and build time of Docker images?*

From a total of 226 changes, we were able to successfully build only 25% of the pre- and post-change snapshot pairs, for a total of 42 changes reducing the size and 15 reducing build time. Two of those instances are in common (*i.e.,* they improve bot size and time). We obtained a relatively high failing rate, mainly related to missing dependencies and, in some cases, missing build ARGs or ENVs. An example is the Dockerfile from `jakowenko/watchtower` repository. Specifically, the commit `0006c06` changes the `node` base image with an alpine flavor.[12] However, the build failed due to a missing package in the `alpine` package repository. The error message was "*unable to select packages: python (no such package): required*

[11] https://github.com/Y-modify/deepl2-infra/commit/0edee8b
[12] https://github.com/jakowenko/watchtower/commit/0006c06

Table 5.1: Summary of the build results for changes reducing the build time.

Repo	*Commit*	*Change*	*p-value*	*Effect Size*	*% Improv.*
`BluezTestBot/action-patchwork-to-pr`	`a44d2cc`	choose a different base image	**0.0001**	large	
`ebimodeling/ghgvcR`	`020fb32`	choose a different base image	**0.0001**	large	
`junron/hwboard`	`1453c38`	remove unused/unnecessary binaries and deps.	**0.0002**	large	
`GiedriusStasiulis/sep6-ng-app-repo`	`a5f566c`	remove layers	**0.0002**	large	
`tlumist/nenn`	`c1a8165`	move dependencies install at top	0.1383		
`briancwy/Imported`	`f70cfee`	move dependencies install at top	0.1314		
`stevesbrain/bitlbee-docker`	`cfcb1e1`	join `RUN`s	**0.0476**	large	
`elixir-cloud-aai/mock-TES`	`020aa0e`	move sources copy at the bottom	1.0000		
`thinca/dockerfile-vim`	`2c5b3ba`	use `wget` instead of `curl` + `git`	0.1852		
`pyrkcommunity/proposal-generator`	`fec50e7`	copy/install requirements beforehand	0.7311		
`politics-rewired/Spoke`	`6bde2df`	use multi-stage build	0.3434		
`uroesch/docker-pa-wine`	`6f9961d`	split `RUN`s	0.4488		
`gbour/wave`	`f9508e1`	split `RUN`s	**0.0012**	large	
`histographer/wizard-backend`	`8faca93`	cache dependencies	**0.0050**	large	

Table 5.2: Summary of the build results for changes reducing the image size.

Change	*Improvement*		
	% Avg.	% Std.	Counts
choose a different base image			1/1
remove unused/unnecessary binaries and deps.		0.17	2/2
use multi-stage build		0.33	10/11
change base image variant		0.27	11/11
remove unused packages			1/1
cache dependencies			1/1
join `RUN`s		0.37	5/6
move dependencies install at top			1/1
yum clean			1/1
remove build/install deps. and sources		0.05	2/3
remove `apt-get` lists		0.13	2/2
remove `apk` cache			1/1
`apk --no-cache`			1/1

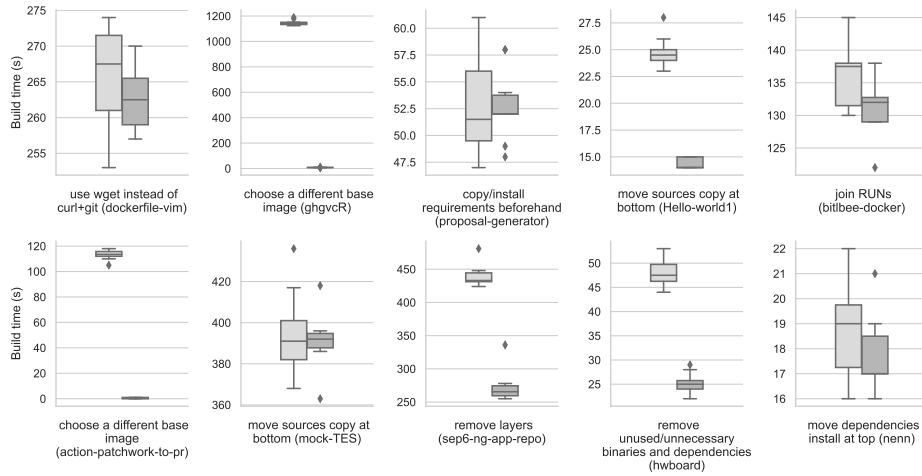


Figure 5.8: Summary boxplots for changes aimed at reducing the image build time.

by: world[python]". Note that the high build failure of previous project snapshots is a well-known problem in the literature, both in the context of CI/CD [118] and, specifically, related to Docker [130]. We excluded from the analysis commit `a2fbb37`[13]. In fact, it resulted in reducing the build time by 10 seconds, on average, but this improvement could not be related to the type of applied change (*i.e.,* move sources copy at the bottom). Therefore, that improvement must come from other factors and operations performed in the Dockerfile, and not the change itself.

Build Time

We report in Table 5.1 the results of the image builds performed for changes aimed at reducing the build time. We also report in Fig. 5.8 the boxplots describing in detail the executions performed to measure the impact on the build time for the changes aimed at its improvement.

The most impacting changes are those aimed at changing the base image with a better one (by almost 100%), along with removing unnecessary dependencies

[13] `https://github.com/rajmohanaravinth/Hello-world1/commit/a2fbb37`

and binaries (by 47%), and moving sources copy at the bottom (by 42%). The same is not true for changes aimed at enabling dependency cache (66% worse) and split of RUN instructions (15% worse), with a *large* effect size. Note that, given our experimental setup, we did not expect to observe a positive impact on the cache (we disabled it). Therefore, those changes might still reduce the build time when the code is changed and Docker caching is enabled. Still, we show that this has a cost on the first build, which takes significantly longer. Commit `8faca93` of the repository `histographer/wizard-backend`[14] provides an interesting example of this: As specified in the commit message, the maven dependencies are cached to reduce the time in case of successive rebuild.

Image Size

We report in Table 5.2 the results of the image builds performed to measure the magnitude of the changes aimed at reducing the image size. For each change, we report the percentage of improvement (*% Avg.* and *% Std.*, measured in MB), excluding the cases in which the image size has been increased. For each type of change, we report the count of those that actually bring an improvement (*Counts*), over all of the evaluated. Also in this case, the changes that allow developers to reduce the image size the most are choosing a different base image (-88%) or a smaller variant (-69%), along with the removal of unnecessary dependencies (-75%). Note that the magnitude of the size reduction due to the change of the base image strongly depends on the specific context (*i.e.,* the previous base image and the current one).

Using multi-stage builds also has a huge impact (-70% size). The same is not true for changes aimed at removing package lists (-11%) and cache (-6%) for `apt`. The change did not reduce the image size in three of the analyzed cases, namely one for *use multi-stage build*, one for *join* RUN*s*, and one for *remove install/build dependencies and sources*.

[14] `https://github.com/histographer/wizard-backend/commit/8faca93`

Summary of RQ_2: Changes aimed at choosing a different base image are the most effective in reducing the final image size (by 88%) and build time (by ∼99%). Removing `apt-get` lists and cache, even if common, has a small impact on image size (by 6-11%).

5.3 Discussion

In this section, we provide some takeaways extracted from our results, together with the implications of our findings.

Finding 1. Choosing an efficient base image is key. Choosing a better *base image* in terms of size or embedded dependencies is the most impacting change to improve performance. In fact, it is the most frequent change performed by developers (106 occurrences), and it allows to obtain a significant reduction of the final image size and build time. A simple rule to select an efficient *base image* is to rely on *official* Docker images, as suggested by the results of Chapter 6. Even better if they embed some of the required dependencies by the application to be contained since it avoids the explicit installation of those dependencies from the Dockerfile reducing the build time. Analyzing more in detail the changes collected in RQ_1, developers usually switch to the *alpine* variant of the same base image, or, in general, they adopt the *alpine* base image.

Finding 2. The cost of the change can be more than the improvement. Some changes attempt to provide an improvement, resulting in being small, but end up degrading the build time or image size since they introduce additional overheads. An interesting case is commit `055a81d`,[15] in which the removal of build and install dependencies increases the build size. This is probably because each added `RUN` instruction produces a layer causing a space wastage. Thus, the space recovered by removing those dependencies is lower than the space occupied by the new layers.

Finding 3. Some changes are useful only for specific usage patterns. Changes like *caching dependencies* or *copy/install requirements beforehand* are not effective in the first build of the image (see the results of RQ_2), but

[15] `https://github.com/rlegrand/dvim/commit/055a81d`

effective in successive builds. This pattern is positive when Dockerfiles are locally used for development, for which it is required to frequently run a build to test the containerized product. However, the contrary is true when they are integrated into CI/CD pipelines that do not rely on caching. The design of our experiment did not target this kind of change. Other types of changes might have a positive impact on one of the aspects we considered and a negative one on the other. This is true, for example, for deciding whether to join or split `RUN` instructions. While joining them can result in a reduction of both built time and image size, the second one can only be useful to reduce the build time when rebuilding the same Dockerfile. This is because, when using the layer caching, only the changed ones are rebuilt while the others are kept in the cache.

Finding 4. Rebuilding past snapshots is not as easy as it should be. The evaluation conducted in RQ_2 gives information about the impact of the changes in real Dockerfiles from open-source repositories. Docker by design aims to make builds reproducible over time. This is not true for cases in which there is poor maintenance (*i.e.,* outdated Dockerfiles [135, 137]) or design issues, such as code smells [13]. Specifically, in our case, we were able to build only 25% of the pre- and post-change snapshot pairs. The literature shows that build fails are commonly diffused in open-source Dockerfiles [130]. This serendipitous finding calls for a more in-depth investigation of the causes of such a phenomenon. A large-scale evaluation is needed to be conducted in this direction. The high failing rate of Dockerfile builds could be mitigated by evaluating existing repairing approaches [45], or else by manually applying those patches.

5.3.1 Implications

A part of the changes reported in our taxonomy (Fig. 5.3) are in line with the best writing practices suggested by Docker [18]. Examples are *using multi-stage build* and *join* `RUN`*s*. Moreover, the existing catalogs of writing violations (*i.e.,* Dockerfile smells) cover also a part of those changes [1, 44]. We report in Table 5.3 the changes identified in our study, compared with existing catalogs and works from the literature. Our investigation proposes 39 new practices. In some cases, the practice is similar to existing ones but extended to a different tool or platform. An example is the usage of `--progress` flag with `npm`, which is

Table 5.3: Summary table of the identified changes that are in overlap with existing catalogs, *i.e.,* Docker Official guidelines [18] (Off), *hadolint* tool [1] (Ha), Binnacle [44] (Bi), `DRIVE` [146] (DR), `DOCKERCLEANER` [9] (DOC), and the study of Ksontini *et al.* [58] (Ks). We reported the cases in which there is a partial (■) or full (✔) overlap.

Change	*Off*	*Ha*	*Bi*	*DR*	*DOC*	*Ks*
`apt-get --no-install-recommends`		✔	✔	✔	✔	
`apk --no-cache`		✔	✔	✔		
remove `yum` cache		✔	✔	✔		
change `ADD` in `COPY`		✔			✔	✔
remove `apt-get` lists		✔	✔			
remove `apk` cache		✔	✔			
remove/avoid `pip` cache		✔	✔			
`pip -no-cache-dir`		✔		✔		
remove `apt` cache		✔		✔		
join `RUN`s		✔				✔
remove install/build dependencies and sources			■	■		
use multi-stage build	✔			✔		
`.dockerignore`	✔					
copy non-mutable sources at beginning	✔					
move dependencies install at top	✔					
move sources copy at bottom	✔					
remove unused/unnecessary binaries and deps.	✔					
remove unused packages	✔					
`apt-get` clean/autoclean		✔				
`npm clean`			✔			
remove temporary files (*e.g.,* `"/tmp/* /var/tmp/*"`)			■			
move installations to external script						✔
remove layers						✔

conceptually similar to rule `DL3047` from *hadolint*. It is worth noting that some of the practices we identified and not present in any previous catalog, like the change of the base image variant, are very frequently adopted by developers and have a substantial impact on either build time or image size.

Thus, by combining our results with existing smell catalogs, we provide a clear direction of what kind of issues researchers and tool builders should focus on. In particular, future research should focus on approaches aimed at suggesting the modifications that require particular effort by developers to apply. An example can be the definition of approaches for the automated refactoring of complex Dockerfiles as multi-stage builds. Also, approaches that can suggest what are the unnecessary dependencies and sources that can be removed, taking as input a Dockerfile. Last but not least, recommending developers a better base image replacement can have a high impact on performance improvement. In this direction, existing approaches for base image recommendation could be adapted to this specific aim [54, 143].

5.4 Threats to validity

In this section, we report the threats to the validity of our study.

Construct Validity. We assumed that the commits selected by the two queries have a concrete impact in terms of build time and image size. However, this could lead to false positives in terms of changes in which the commit message reports the intention of an improvement, but the change itself is not able practically to provide any improvements. An example is commit `bb96380` from `cropgeeks/docker`[16]: It is not effective in reducing the build time of the image (*i.e.,* it will result in a negligible improvement). The manual validation step performed by the annotators, and the subsequent discussion with a third author, allowed to avoid those cases. The same is true for the presence of false negatives: The only measure we could take to mitigate this threat is to design simple and generic queries to get the most out of the commit of interest.

Internal Validity. We used two different queries when we extracted the commits. However, they can seem to be very simple (*i.e.,* only keyword match-

[16] https://github.com/alubbock/thunor-web/commit/c77ccbb

ing) and lead to false positives. We avoided using complex queries to avoid sample bias in the commit selection. Also, we manually validated the selected commits to remove any false positives. There is a possible subjectiveness bias introduced during the manual annotation of the change applied by each commit. We mitigated this by adopting a conservative approach, *i.e.,* we did not "interpret" the commit change, but we mainly relied on information provided by the commit message. Also, the process has been executed independently by two different annotators discussing and resolving conflicting tags.

External validity. The taxonomy proposed in our study is based mainly on open-source Dockerfiles. This means that there could be some differences when applied in an industrial context. As a plus, the practices that are captured in our taxonomy cover some of those suggested in the official Docker guidelines [18] and as code smells [1]. This means that our findings are an enrichment of the existing practices.

5.5 Final Remarks

Optimizing the resources used by Dockerfiles and Docker images is one of the most important aspects in which developers invest their effort. In this chapter, we presented an in-depth empirical evaluation of the changes performed by developers in the open-source to improve the performance of Docker images in terms of image size and build time. First, we extracted a set of changes, reducing the build time and image size, from open-source git repositories. We selected them by combining NLP techniques and manual validation to exclude false positives. Then, we tagged the applied operations aimed at reducing build time and image size and we grouped them in a taxonomy of changes. We quantified and reported the practical impact by building pairs of Dockerfile snapshots *pre-* and *after-* those changes were applied.

In summary, the takeaways of this chapter are the following:

- Developers mainly reduce the build time and image size of Dockerfiles by removing unnecessary dependencies and binaries, switching the base image with a more efficient one, and optimizing the usage of cache;

- The change impacting the mos by reducing both image size (by 88%) and build time (by ∼99%) is the replacement of the base image with a more efficient one;
- In some cases, the cost of the change is more than the improvement it provides (*e.g.,* removing build and install dependencies);
- Changes aimed at enabling caching and optimizing the instruction order are not effective in the first build of the image, but they improve the successive re-builds.

A future contribution, in the same direction, could be a large-scale study aimed at measuring the impact of the changes we found on both image size and build time. Such a study is difficult to conduct due to the high failing rate of existing Dockerfile snapshots. This challenge could be mitigated by (i) adopting approaches for the automated repair of Dockerfiles, such as Shipwright [45], or (ii) manually fixing the issues in failing Dockerfiles.

Acknowledgments

We thank the student Mattia Iannone (University of Molise, Italy) for his precious contribution to the development of the commit selection heuristic and the manual validation of the selected commits.

CHAPTER 6

Understanding What Quality Aspects Characterize the Adoption of Docker Images

When writing the Dockerfile for a given application, developers usually start from a pre-existing image containing the basic dependencies needed. For example, to containerize a Java application, it will be necessary to provide the Java Runtime Environment (JRE): Therefore, a base image with the JRE could be adopted. However, many alternative images exist that provide the same (or analogous) dependencies, and developers find it difficult to search for Docker images on DockerHub [8, 51]. As we discussed previously, several studies in the literature investigated different types of quality features and metrics for Docker images. In general, we can group those features into two distinct groups. The first is composed of *i.e., externally observable features*, which influence their adoption as they are what developers and image users can observe when they have to choose a Docker image to use. Such features include, for example, the image size [58] related to the resources that the image will use, and the presence of software vulnerabilities [109, 73, 137] which can lead security risks. Such *externally observable features* are influenced by a second group of features, namely *configuration-related*

features. Those features describe related development aspects of the Dockerfiles that might positively or negatively affect the resulting Docker image. Examples are the presence of Dockerfile smells [129, 66] which can lead to the introduction of security issues [137].

Static analysis tools can support developers to follow best practices in Dockerfiles [1, 44, 132] and, thus, minimize the presence of *internal* quality issues. However, they may not be sufficient to assess the absence of code smells [70]. Despite such exemplary features and the presence of a plethora of studies that focus on specific quality issues, the literature lacks a general view of what are the externally observable and configuration-related features of Docker images and Dockerfiles. Similarly, to the best of our knowledge, it is unclear (i) how externally observable features impact developers' preferences when they have to choose a Docker image, and (ii) the impact of configuration-related features on the *external* ones.

In this chapter, we aim to fill these gaps. First, we reviewed 31 papers to define a comprehensive taxonomy of externally observable features and configuration-related features features of Docker images and Dockerfiles. Then, we conduct an empirical study on a dataset of 2,441 open-source Docker images. We aim at finding out what *external* features impact the developers' preferences in terms of actual *adoptions* (*i.e.,* how frequently they appear in the `FROM` statements of app-specific Dockerfiles) and perceived quality, intended as the *prominence* of a Docker image over others (*i.e.,* number of stars on DockerHub). Our results show that, as expected, official Docker images have a positive relationship with both *adoptions* and *prominence*. Besides, both image size and the number of exposed secrets (*i.e.,* a metric related to security) negatively impact the developers' preferences. Interestingly, the number of vulnerabilities only impacts the prominence of the image, but not the actual number of adoptions. This result suggests that developers are aware that some problems affect the quality of the images, but this does not change their behavior when they have to choose a Docker image to use (mostly because they are not aware of alternatives [51]). Moreover, our results show that the less the number of SLOC, the less the occurrence of vulnerabilities as also shown in previous studies [4, 92]. In the same way, the image size decreases when the number of LOCs decreases. This means that a smaller

image size has a positive impact on the developers' preferences. Also, we found no relationship between the presence of Dockerfile smells and any of the external features. Shell script smells, instead, have an impact on security-related features. However, there are some exceptions. This is mainly because, as we performed a correlation study, it can not be implied causality based on these results. For example, not all instructions (in terms of SLOC) directly impact the image size. This not apply when removing instructions like `EXPOSE` or `LABEL`. On the other hand, shell script smells are not always related to security. It is proven that mature Docker images tend to have fewer security issues [109], despite the number of smells.

To summarize, we provide the following contributions:

- We define a taxonomy of metrics and attributes extracted from a total of 31 research papers through a literature review;
- We conduct an empirical study on a total of 2,441 Docker images to evaluate which external features developers consider important in terms of adoption or to positively evaluate Docker images;
- We find out what are the configuration-related features that affect the externally observable features that are related to the adoption of Docker images.

The rest of the chapter is organized as follows. In Section 6.1 we describe the procedure used for building the taxonomy of features and metrics of Dockerfile artifacts. In Section 6.2 we present some hypotheses related to the impact of the quality features on developers' preferences. In Section 6.3, we present our empirical study to evaluate the impact of the quality features on the developers' preferences. We discuss the results in Section 6.4, and the threats of validity in Section 6.5. Finally, in Section 6.6 we provide final remarks and future directions.

Table 6.1: Inclusion and exclusion criteria for the selection of primary studies.

Inclusion Criteria	
IC1	The paper has been peer-reviewed (published either in a journal or in the proceedings of a conference)
IC2	The elements treated are either Docker images or Dockerfiles
IC3	The paper title or abstract contains the keywords *quality* and *Docker* in the title, or is explicitly referenced by another paper matching this criterion and contains quality-related keywords (*e.g.,* refactoring, smell, bug)
IC4	The paper focuses on non-functional aspects of Docker images or Dockerfiles related to quality
Exclusion Criteria	
EC1	The paper is not written in English language
EC2	The paper is not published by IEEE, ACM, Springer, Elsevier
EC3	The paper focuses on aspects related to the architecture of Docker images (*e.g.,* storage system)
EC4	The paper is not presenting quality metrics for Docker images or Dockerfiles
EC5	The paper is not a technical article published in a journal or in the proceedings of an international conference/workshop

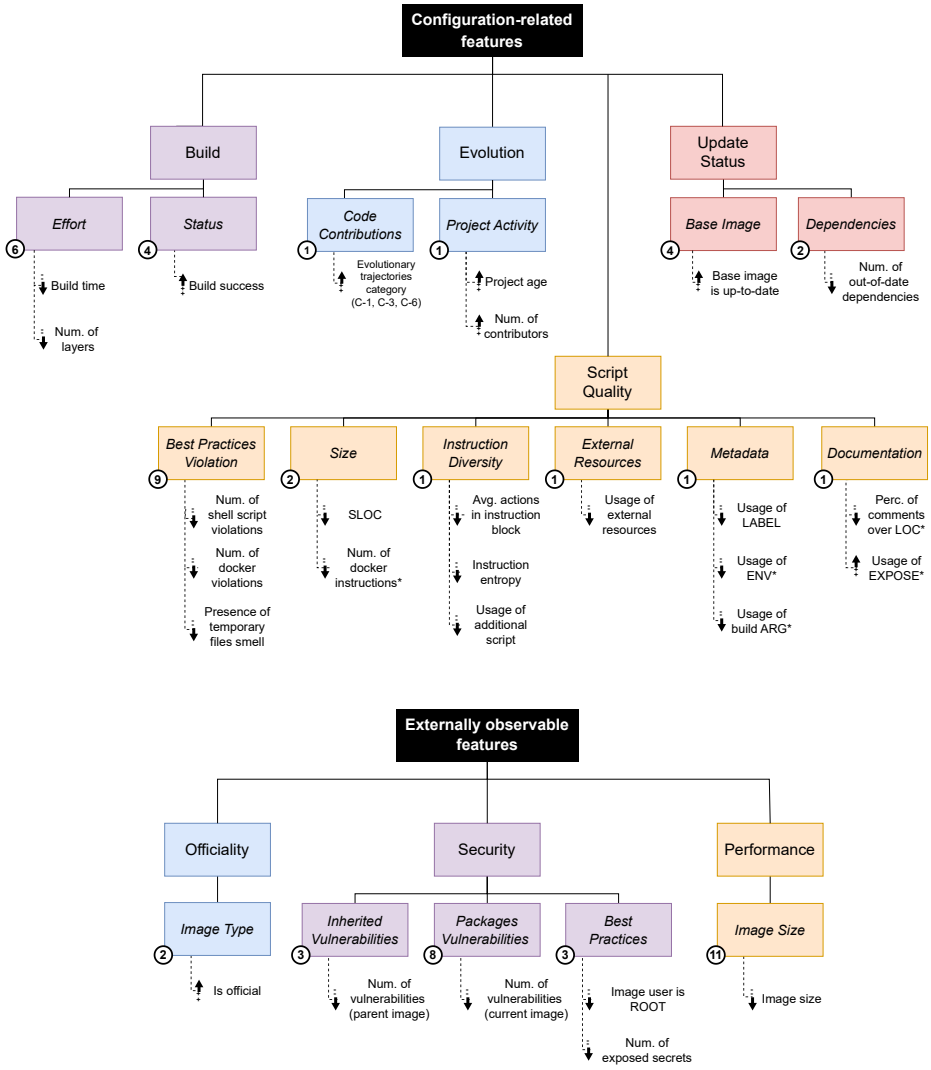
6.1 Discovering External and Configuration Features of Docker Artifacts

In this section, we present the preliminary study we conducted to collect the quality features and metrics of Docker images and Dockerfiles. We first present the methodology we used for collecting and analyzing relevant papers on Dockerfile quality, from which we aim at extracting knowledge, and then we present the obtained results.

6.1.1 Methodology

The *goal* of this preliminary study is to collect a set of configuration-related features and externally observable features of Docker images and Dockerfiles. To achieve this, we conduct a literature review of scientific articles about Docker quality, and we qualitatively analyze them to extract the information related to features and metrics. We have not performed a rigorous Systematic Literature

Figure 6.1: Taxonomy of external and configuration features and metrics. For each feature, the number of references from the literature is reported. For each metric, an up or down arrow indicates if it is positively or negatively correlated to the feature it measures.



Review (SLR) on quality aspects because the topic is too broad, and it would have been outside the scope of this step (*i.e.,* selecting quality metrics). We describe below, in detail, the procedure we followed.

Identification of Relevant Articles

We searched for studies regarding Docker quality, as a general topic. To do this, we relied on Google Scholar, and we used the generic query *"docker quality"*. We collected a core set of articles that conduct studies on the quality of Docker images and Dockerfiles. Specifically, starting from the first paper returned by Google Scholar, we considered all subsequent papers stopping when the title and abstract did not contain the keywords *docker* and *quality* ($\sim$ 30 results). We defined a set of inclusion and exclusion criteria, reported in Table 6.1, for selecting the articles of interest. After having collected the first set of papers, we read their titles and abstracts, and we verified the criteria *IC1*, *IC2*, *IC3*, *EC1*, *EC2*, *EC5*. At this stage, if we were not sure whether any of the used criteria were met, we kept the paper. Next, we used snowballing (*i.e.,* we analyzed the relevant references of the selected papers) and looked for more recent papers citing them by relying on, again, Google Scholar. We used the previously described process to filter them and include, in the end, only the possibly relevant ones. We applied a less strict filter on the title and abstract, also looking for words related to quality improvements (*e.g.,* refactoring, technical debt, repair) or quality-related aspects (*e.g.,* smells, build failures, security, performance, bugs). Finally, we carefully read the whole papers and filtered them using all the inclusion and exclusion criteria. In total, we analyzed 75 articles. We excluded 44 of them, and we were left with a total of 31 relevant articles to analyze in the next steps.

In terms of editorial collocation, most of the papers we selected were published in the proceedings of international conferences (*i.e.,* 23 of 31) while only 7 of them were published in journals. The most occurring venue is *Mining Software Repositories (MSR)*, with 5 articles, followed by *International Conference on Software Engineering (ICSE)* (4 articles). The temporal collocation is between 2017 and 2022, and most of the articles are from 2019 (16 out of 31). This is expected, given the fact that Docker was introduced in 2013 and, therefore, the scientific interest in the adoption of such a tool has started increasing only

recently, following the adoption by developers of open-source software, for which data are easily accessible.

Qualitative Analysis Methodology

We analyzed the selected articles to find out the discussed metrics and features related to quality from the literature. For extracting the information of interest, we adopted the card sorting approach [113]. We identified, for each paper, the quality *features* and the possible *metrics* defined to measure them. Two of the authors, independently, assigned one or more tags to each article by distinguishing tags related to the quality *features* and the ones related to the quality *metrics*. Given the set of assigned tags for each category (*features* and *metrics*), we analyzed them, aiming at using a unique expression when the two evaluators used different tags for expressing the same concept (*e.g.,* "image size" and "size"). The two evaluators discussed the cases in which there were conflicts on the assigned tags, aiming at reaching a consensus.

After having completed the tag assignment, we organized the tags related to the quality features in a first version of the taxonomy. Then, we added to the taxonomy, as children of the leaf features, all the tags related to the metrics we identified for such a feature.

The taxonomy is divided into two parts: externally observable features, *i.e.,* what image users can observe, and configuration-related features, *i.e.,* aspects related to Dockerfiles and the build process of Docker images. The former, mainly measured on the Docker image itself, is what the adopters of the image (*i.e.,* artifact) immediately can see from DockerHub or from the image metadata. The latter are mainly measured by analyzing the Dockerfile, which is what the developers primarily see (*i.e.,* source code), or related to the build process which involves both the Dockerfile and the image (*e.g.,* build time). We assigned an up or down arrow to report, for each metric, if it is positively or negatively correlated to the measured feature.

Table 6.2: Summary table of Docker configuration-related and externally observable metrics. The symbol ***** means that is a newly introduced metric.

	Metric	Rationale	How to compute	Reference
Configuration-related features	*Build time*	Build time of the Docker image	Build time measured from the execution time of the `docker build` command	[66, 44, 144, 142, 48, 58]
	Num. of layers	Build effort of the Docker image	Number of layers that compose the Docker image, using `docker history` command	[142]
	Build success	Build status	`TRUE` if the build was successful completed, `FALSE` o/w	[13, 130, 44, 45]
	Evolutionary trajectories category	Evolution of Dockerfile over time	Clustering of Dockerfile evolutionary vectors, computed using changes history [141]	[142]
	Project age	Evolution/Project activity	Time passed between first and last commit, measured in seconds	[129]
	Num. of contributors	Evolution/Project activity	Number of contributors that made at least one commit to the project repository	[129]
	Num. of shell script smells	Presence of code smells	Number of Docker-related violations (DL-XXXX) detected by *hadolint* tool	[129, 66, 43, 44, 27, 142, 45, 58, 5]
	Num. of docker smells	Presence of code smells	Number of Shell-related violations (SC-XXXX) detected by *hadolint* tool	[129, 66, 43, 44, 27, 142, 45, 58, 5]
	Presence of temporary files smell	Presence of code smells	Semi-automatic approach as described by Lu *et al.* [70]	[70]
	SLOC	Size of the Dockerfile	Number of lines of code (LOC), excluding code comments	[129, 142]
	Num. of docker instructions	Dockerfile complexity	Total number of the Docker instructions (*e.g.,* `RUN`, `COPY`, etc.) used in the Dockerfile	[129, 142]*
	Layer size	Dockerfile complexity	Average number of concatenated commands per instruction block (separated by &&)	[142]
	Instructions entropy	Dockerfile complexity	Shannon entropy computed among the different (unique) Docker instruction used in the Dockerfile	[142]
	Usage of additional script	Usage of additional resources	`TRUE` if the Dockerfile contains a shell script execution (*e.g.,* running bash scripts ending with `.sh`), `FALSE` o/w	[142]
	Usage of external resources	Usage of additional resources	`TRUE` if `wget` or `curl` commands calling external URLs are used in the Dockerfile, `FALSE` o/w	[142]
	Usage of LABEL	Usage of instructions for image metadata	`TRUE` if the `LABEL` instruction is used in the Dockerfile, `FALSE` o/w	[142]
	Usage of ENV	Usage of instructions for image configuration	`TRUE` if the `ENG` instruction is used in the Dockerfile, `FALSE` o/w	[142]*
	Usage of build ARG	Usage of instructions for image configuration	`TRUE` if the `ARG` instruction is used in the Dockerfile, `FALSE` o/w	[142]*
	Perc. of comments over LOC	Presence of code documentation	Percentage of number of comments over SLOC, computed as: *n. comments*/*SLOC*	[142]*
	Usage of EXPOSE	Presence of code documentation	`TRUE` if the `EXPOSE` instruction is used in the Dockerfile, `FALSE` o/w	[142]*
	Is base image up-to-date	Update status	Checking the Docker image repository on DockerHub for image tag updates	[109, 66, 54, 58]
	Num. of out-of-date dependencies	Update status	Checking the software packages repository for the presence of updates running `apt update` (for Debian-based images)	[137, 136]
Externally observable features	*Is official*	Vendor of the Docker image	`TRUE` if the Docker image has the *Official Image* badge in *DockerHub*	[129, 34, 142]
	Image size	Size of the Docker image	Image size measured in bytes, computed by adding up the size of all the image layers obtained from the `docker history` command	[70, 132, 66, 114, 44, 144, 142, 145, 92, 111, 58, 5]
	Num. of vulnerabilities (parent image)	Security vulnerabilities	Number of security vulnerabilities (v) considering only the base image (BI) using *Clair* scanner (BI_v)	[109, 67, 92]
	Num. of vulnerabilities (current image)	Security vulnerabilities	Number of security vulnerabilities (v) introduced by packages added in the resulting image (RI), using *Clair* scanner ($RI_v - BI_v$)	[73, 137, 66, 114, 135, 136, 54, 51]
	Image user is root	Security best practices	`TRUE` if the Docker image runs as root-enabled container, `FALSE` o/w (*e.g.,* using the *whaler* tool)	[73, 67]
	Num. of exposed secrets	Security best practices	Number of exposed login credentials or access tokens detected in the Docker image (*e.g.,* using the *whaler* tool)	[109]

6.1.2 Taxonomy of Quality Features and Metrics

The resulting taxonomy is described in Fig. 6.1. The boxes with italicized text indicate the features, while the others indicate categories of features we introduced in the taxonomy. Also, in Table 6.2, we report the quality metrics and the papers resulting from the literature review. The numbers in the circular badges, instead, indicate the number of papers that use the feature. Next, we describe the categories we identified for both configuration and external features.

Configuration-Related Features

Configuration features are all the features related to the Dockerfiles behind the Docker images and the build procedure which involves both the artifacts. Such features are not directly perceived by the users of a Docker image, similar to how internal code quality aspects (*e.g.,* the maintainability of a software system) are not directly perceived by the end users. However, they are important for the Dockerfile developers, and they might eventually impact some of the externally observable features which are, instead, directly perceived by the users. We identified the following categories:

Build. With this category, we indicate the aspects related to the build process of the Dockerfile. A slow build, for example, increases the time needed to update the software in production if continuous deployment is adopted. The *Effort* feature represents the resources involved in the build process (*e.g.,* time) [142], while the *Status* feature indicates the success or failure of the build process (*i.e.,* if Docker image builds or not) [130].

Evolution. This category embraces the aspects that are related to the evolution of the Dockerfile. The *Code Contributions* feature indicates the modifications made to the Dockerfile in time. The *Project Activity* feature, instead, describes the aspects related to the development process, such as team composition. Large development teams may be better at writing good quality Dockerfiles (*i.e.,* more technical knowledge) [129].

Script Quality. This category contains all the features strictly related to the quality of the source code. The feature *Violation of Best Practices* represents the presence of Dockerfile smells [129]. The feature *Dockerfile Size* represents the as-

pects related to the size of a Dockerfile, such as the number of lines of code. The *Instruction Diversity* feature is related to the homogeneity of the source code: A more heterogeneous code (*i.e.,* source code that has many different instructions) can lead to misleading developers [142]. The *External Resources* feature regards the usage of resources not provided in the original project repository, such as libraries or other files downloaded from remote servers [142]. The feature *Metadata* describes the use of meta-data in the Dockerfile, such as environment variables or the `LABEL` instruction [142]. Finally, the feature *Documentation* describes the use of documentation in the Dockerfile [142]. Code comments are an example of documentation. If the script quality of the Dockerfile is low, it is intuitively more likely that different kinds of issues arise (*e.g.,* security-related) given the lower maintainability [129, 66].

Update Status. This last category contains the features that are related to the maintenance status of a Dockerfile. The feature *Base Image* captures the update status of the Docker image used as a base of the Dockerfile. On the other hand, the feature *Dependencies* is about the updated status of additional software packages used in the Dockerfile. If a Dockerfile is not maintained, it is more likely that some of the dependencies are out-of-date, and this might negatively impact the security of the whole image.

Externally Observable Features

The external features are related to Docker images, the software artifacts that derive from a Dockerfile after the build process. Such aspects might be directly perceived by developers who use the image if, for example, they adopt it as a base image. We identified the following categories of features:

Officiality. With this first category, we indicate the degree of officiality of the image or of the developer(s) who published it. It is reasonable to assume that official images, or images published by trusted developers, are perceived better by developers because they are preferred over unofficial ones [34].

Performance. The way in which Docker images use the available resources might be crucial for developers since it also impacts the cost of operation. *Image Size*, specifically, is the only relevant feature related to this category, and it

indicates the storage needed to use the image. Developers tend to dislike images bigger than necessary (*e.g.,* if they contain unnecessary software packages) [66].

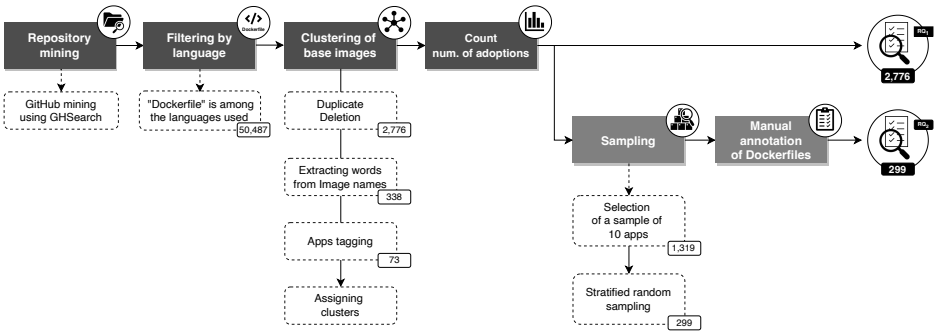
Security. We include, in this last category, all the security-related aspects of a Docker image. The *Best Practice* feature concerns the adoption of the main security best practices of a Docker image. An example of best practice in terms of security is the usage of a user different from root, as the default user, when the image is executed. The *Inherited Vulnerabilities* and *Packages Vulnerabilities* features are related to the number of security vulnerabilities found in the image based on the *Common Vulnerabilities and Exposures* (CVE) database. The first one only concerns the parent image of the actual Docker image (*i.e.,* the base image used in the Dockerfile), while the second one concerns the additional software installed in the image. Developers must prefer images that provide all the necessary security-related features, to avoid security risks [67, 73].

Metrics

Table 6.2 describes in detail the metrics defined in our taxonomy and the features that they aim at capturing. While most of them were already defined in the papers we analyzed, we introduced some new metrics and variations of existing ones to better measure some of the features that compose our taxonomy. We describe below only the differences with respect to the existing ones, which are summarized in Table 6.2.

Configuration-related features. We introduced *Num. of docker instructions*, a new metric for measuring the *Size* of a Dockerfile. Such a metric counts the number of Docker instructions in the Dockerfile. Since each instruction of a Dockerfile will be converted to an image layer, a Dockerfile having many instructions will generally have a higher number of layers. It is worth noting that the number of instructions might be lower than the LOCs since a single instruction might encompass many lines. For the feature *Metadata*, we defined two more metrics: *Usage of ENV*, which measures the number of environment variables used in the Dockerfile, and *Usage of build ARG*, which measures the number of build arguments. Such metrics are inspired by *Usage of LABEL* [142], which indicates the presence of the `LABEL` instruction in Dockerfiles. The metrics *Perc. of comments over LOC* and *Usage of EXPOSE* are additional measures

Figure 6.2: Dataset extraction procedure. The labels at the bottom show the number of selected instances up to that step.



for the *Documentation* feature. The first one is a variation of a metric defined by Zhang *et al.* [142]: While the original version measures the absolute number of comments, our metric computes the percentage ratio between the number of comments and LOC. It is expected that the ratio, more than the absolute number of comments, is important to determine to what extent the Dockerfile is well-documented. Usually, developers tend to give an explanation comment of what each instruction does [5]. The last metric we introduced, *i.e., Usage of EXPOSE*, is boolean, and it checks the presence of the `EXPOSE` instruction. Such an instruction has the purpose of documenting the ports to be used when the Docker container will be executed.[1]

Externally observable features. We defined two new metrics for the *Security/Best Practices* feature, *i.e., Image user is root* and *Num. of exposed secrets*. *Image user is root* is a binary metric that indicates whether the principal user of the image is root or not: A good security practice, indeed, is to use containers for which the main user does not have root privileges (*i.e.,* non-root user). *Num. of exposed secrets* measures the estimated number of secrets (*e.g.,* passwords or private keys) stored in the image: A good security practice is to avoid exposing sensitive data [109]. Therefore, the lower such a metric, the higher the security.

[1] `https://docs.docker.com/engine/reference/builder/\#expose`

6.2 Explaining Developers' Preferences

Software developers implicitly or explicitly express their preferences on Dockerfiles in several ways. They can do it *explicitly*, by starring the Docker image on DockerHub, or *implicitly*, by adopting the image in their own Dockerfiles. In both cases, we hypotize that the *external* features we identified from the literature in the previous section influence the developers' preferences. Specifically, we formulate the following *hypotheses*:

Hypothesis 1. Developers prefer images with fewer security issues.

We expect that developers are, to some extent, aware of the security issues of the images they use and, therefore, they prefer alternatives that do not have security issues (or that, in general, have fewer of them).

Hypothesis 2. Developers prefer smaller images.

We expect that developers prefer Docker images that, by offering the same features (*i.e.,* installed software and dependencies), use a lower amount of space.

Hypothesis 3. Developers prefer official images.

We expect that developers prefer official images over non-official ones since they are guaranteed to provide a minimum quality level.

We also hypotize that configuration features related to the Dockerfiles influence external features. Developers that use Docker images do not directly perceive configuration features (*e.g.,* they are not aware of the LOCs of the Dockerfile). Therefore, we assume that configuration features only have an indirect influence on the developers' preferences. Specifically, we formulate the following *hypotheses*:

Hypothesis 4. The number of layers and the adoption of bad practices increase the size of a Docker image.

We expect that features related to the build effort and script quality are correlated with an increase in the final Docker image size on disk. The composition of a Docker image (*i.e.,* layers) is directly related to the build effort in terms of resource usage. Fewer layers might be related to both less build latency and less storage used. Besides, we expect that a Dockerfile written following best practices can produce a more optimized in terms of resources since some best practices are precisely aimed at this.

Hypothesis 5. The complexity of a Docker image and bad practices in its development process increase the number of security issues.

A complex Docker image might result in low Dockerfile quality. Thus, we expect that a more complex Docker image leads to a higher number of security issues (among other issues), as it has been observed for normal source code [69]. Complexity metrics are related to the presence of security vulnerabilities, together with the developers' activity (*e.g.,* team size) [108]. Thus, we also expect that bad practices in the development process can increase security risks in the Docker images.

We do not formulate *hypotheses* regarding the officiality of the Docker image, since the process behind the assignment of the "official image" badge is well-known [19].

6.3 Empirical Study Design

The *goal* of the study is to understand which external features directly influence the developers' preferences and which configuration features indirectly do so (by directly influencing external features). The *context* consists in 2,441 open-source Docker images used as *base images* for 10 software applications hosted

on GitHub, and on 299 Dockerfiles manually associated to a sample of Docker images from 2,441.
Our study is steered by the following research questions:

RQ_1: *Can the externally observable features explain the developers' preference for a Docker image?* With this first research question, we want to know what external features, *i.e.,* those related to the Docker image, allow to explain the adoption and the preference expressed by the developers in terms of adoptions (how many times a Docker image is used as a base image in Dockerfiles) and perceived quality (prominence measured as the number of stars on DockerHub). This research question will allow to verify or disprove *hypotheses* 1, 2, and 3.

RQ_2: *Are configuration-related features correlated with externally observable features for Docker images?* With the second research question, we want to understand which configuration features directly influence the external features of a Docker image and, thus, indirectly influence the developers' preferences. This research question will allow to verify or disprove *hypotheses* 4 and 5.

6.3.1 Data Collection

The context of our study is composed of *objects*, *i.e.,* Docker images and their related Dockerfiles. In our study, we built two distinct datasets from the open-source codebase: D_{img} and D_{src}. D_{img} is composed of 2,441 instances of Docker images, associated with the respective number of adoption and the number of DockerHub stars. D_{src} contains a subset of the images from D_{img} (299) manually associated with the Dockerfiles used to build them. We use D_{img} for answering RQ_1 and D_{src} for RQ_2. The procedure we used for building such datasets is summarized in Fig. 6.2 and detailed below.

Dataset of Docker Images and Developers' Preferences (D_{img})

Mining Adoptions of Docker images Our main objective with D_{img} is to annotate a set of Docker Images with their number of adoptions in downstream

Dockerfiles and DockerHub stars. While the latter can be easily achieved by using DockerHub APIs, the former requires mining existing software repositories. To do this, we use *GHSearch* [17], which crawls data from open-source software projects hosted on GitHub providing metadata and statistics such as commits, contributors, stargazers and the other information related to the repository. We extracted the metadata for GitHub project repositories, as provided by the tool, starting from the date when Docker is introduced, *i.e.,* 2013, to January 2022. Next, we selected only the repositories where "Dockerfile" is among the language used to exclude projects that do not use Docker. As a result, we obtained a total of 50,487 projects. Then, we collected all the Dockerfiles from such projects (182,375, in total) and we extracted their content at the latest snapshot. We parse the Dockerfiles obtained, and we extract all the base images used (*i.e.,* the ones which follow the `FROM` instructions). As a result, we obtained a list of base images used. Finally, we get the unique images, and we count, for each of them, how many times they occurred. The final result is a set of 20,425 Docker images used as base images associated with the respective number of adoption (*i.e.,* how many times they appear in the `FROM` instructions).

Annotating Docker Images with Application Besides having the number of adoption for the collected Docker Images, we also want to annotate them with the software they provide and its version. This is necessary because, to answer RQ_1, we will need to group together all the images providing the same features and explain the developers' preferences among them, rather than among images providing different features. Indeed, let us imagine that we have two Docker images providing an Apache HTTP server, with 1,000 and 900 adoptions, respectively, and an image providing Nginx, with 2,000 adoptions. We do not know whether the higher number of adoptions is due to the fact that developers prefer the Docker image providing Nginx or they simply prefer Nginx. In other words, the number of adoptions between the two images providing Apache HTTP is comparable and might depend on the differences between the images, while the number of adoptions of the image providing Nginx can not be mixed with the others. The same is true for different versions of the same software: Developers might prefer a given version of Apache HTTP and base the choice on it rather

than on the non-functional aspects of the Docker image. Therefore, we assigned each image with an *application name* (*e.g.,* "Tomcat") and an *application version* (*e.g.,* 7.0). To do this, we use a semi-automatic procedure. First, we removed all the instances where the Docker image repository name contains special characters that are not allowed by the Docker naming convention (*i.e.,* non-alphanumeric symbols or placeholders). Thus, from a total of 182,375 instances, we retain 141,583 of them. Next, we extracted the words contained in the image names by performing a string split over the separators (*i.e.,* dash or underscore). For example, from *alpine-maven-builder-jdk-8*, we extract the words *alpine*, *maven*, *builder*, *jdk* and *8*. The next step is to select, among all the obtained words, only those that are alphabetic (*i.e.,* do not contain symbols or numbers) and contain at least 3 characters. We do this to discard words that are not useful. Examples are *go*, *os*, *js* as we select Docker images containing applications and not OSes and programming languages. We selected all the words appearing in at least five image names, and we obtained a total of 338 unique words. Each of the selected words is a candidate application name. We discard word (*i.e.,* candidate applications) with less than five occurrences to avoid having too small groups for the analysis performed in RQ_1 and include software that is provided through a limited number of Docker images.

Next, we selected and assigned a set of tags (*i.e.,* clusters) to group each base image of our dataset by the contained application. For example, we assign the label *tomcat* to all the images that provide the tomcat web server. We used the dataset of Docker images obtained in the previous step to achieve this. At this point, a manual process is required to identify if a word corresponds to an application name to group similar Docker images (*i.e.,* clustering). This is done by manual annotation of all the extracted words that occur at least 5 times, *i.e.,* there are at least 5 unique Docker images containing those words, for a total of 338. Then, we manually check the candidate application names, and we select only the ones that are actual applications. We discard operating systems/Linux distributions (*e.g., ubuntu*, *debian*, *alpine*), programming languages (*e.g.,* python, java), and other commonly used words which do not pertain the application (*e.g., build*, *base*, *dev*, *runtime*, *aws*, *platform*). Examples of valid words we selected are *nginx*, *maven*, *jenkins*, *chrome*, *dotnet*, *envoy*, *mysql*. In some cases,

different words could refer to the same application (*e.g., postgres* and *postgresql*). In such cases, we manually created clusters of names and associated them with a unique name (*e.g., postgres*, in the previous example). As a result, we obtain a total of 73 different applications associated with all names through which they appear in the Docker images. Finally, we associated each Docker image with a list of applications it provides by simply performing string matching with the words analyzed in the previous step. If a Docker image was associated with no application, we discarded it. This happened, for example, for Docker images providing Linux distributions, as previously explained. We manually analyzed cases in which a Docker image was associated with more than an application, and discarded the cases in which more than an application was actually provided. After this step, we obtain our final dataset of 2,776 Docker images (covering a total of 12,674 adoptions). We also annotate each image with the version of the application provided. To do this, we split the Docker image name as previously done to identify the application name, and we select the word with the highest number of numeric characters. We manually check if the version assigned to each image was correct.

Feature Extraction We added to the dataset all the features needed to answer our research questions. Firstly, for each Docker image, we extracted the number of stargazers (*i.e.,* stars) by using the DockerHub APIs,[2] to compute the perceived quality (*i.e.,* the prominence of a Docker image over the others, used as a dependent variable for RQ_1). We computed most of the metrics related to the external features from the literature we identified in Section 6.1.2, with some exceptions and small variations. We describe below only such cases. We did not consider the metric *Presence of temporary files smell*, because it can not be exactly measured automatically but only with a semi-automatic approach, as described in the reference article [70]. Moreover, we merged the metrics for the feature *Inherited Vulnerabilities* and *Packages Vulnerabilities* in *Num. of vulnerabilities*. We did this because, given a Docker image, we could not distinguish the layers inherited from the base images (*i.e.,* parent) and the additional layer added on top of them with the specific Dockerfile used, since we do not have

[2] `https://docs.docker.com/docker-hub/api/latest/`

such a Dockerfile in D_{img}. To compute the *Num. of vulnerabilities*, we used the *Clair* tool. For the metrics in the category *Security/Best Practices*, we adopt the *Whaler* tool, which returns *Image user is root* and *Num. of exposed secrets*. To measure the *Officiality* feature, we implemented a web scraper to parse the presence of the label *"Official Image"* on DockerHub.

Table 6.3: Summary of the selected applications and sampled instances from the dataset.

Application	**Instances**	**Sample**
Nginx	344	78
Cuda	229	52
Maven	177	40
Tomcat	147	33
Postgres	143	32
Redis	79	18
Elasticsearch	65	15
MySQL	65	15
fluentd	58	13
Dotnet	12	3
Total	1,319	299

Dataset of Dockerfiles associated with Docker Images (D_{src})

To perform the analysis required in the context of RQ_2, we need to have, for each Docker image, the source Dockerfile. Thus, we defined a second dataset, namely D_{src}, which contains a subset of the Docker images from D_{img}, in which each instance contains the content of the Dockerfile used to build it. To achieve this, we first randomly extracted a sample of D_{img} for the applications with the highest number of Docker images. We filtered D_{img} and selected only the Docker images for such selected applications obtaining a total of 299 instances. Manually annotating the Dockerfile from a Docker image is challenging: In most cases, a direct link to the Dockerfile is missing. Thus, we performed a random sampling selecting 299 total instances with a confidence level of 95% and 5% margin of

error. Finally, we manually annotated the Dockerfiles related to each remaining Docker image. To achieve this, for each image, we looked at the DockerHub repository. If there was a direct reference to the Dockerfile, we assumed it was the one used to build it. Otherwise, we performed a Google search using the name of the image plus the word "Dockerfile" (*e.g., nginx Dockerfile*) looking for the source of the Dockerfile related to that image. If we obtained no results, we replaced the Docker image with another randomly selected, for the same application, to avoid hampering the representativeness of our sample. We report in Table 6.3 the total number of selected applications and the sampled number of instances, *i.e.,* the different groups of comparable Docker images and their number, involved in our experiment. In detail, we have 10 different groups having a number of Docker images varying from 12 (dotnet) to 344 (nginx). We have a total number of 2,441 open-source Docker images, and for a subset of them (299) we also have the source Dockerfile from open-source codebases.

Also in this case, we computed on D_{src} all the metrics related to the configuration features that were reported in Section 6.1.2, with some exceptions and small variations. We describe below only such cases. We excluded from the metrics related to the feature *Update Status* because we could not have a reliable measure for the metrics *Is base image up-to-date* and *Num. of out-of-date dependencies*. The update status of the base image and the package dependencies, indeed, depends on the time at which the adoption was made in the downstream Dockerfiles, and it changes over time. We cannot trace back the time at which one or more dependencies (possibly) became out-of-date in a Docker image and, thus, report if it was so at the time of adoption. Also, we do not compute the metric *Evolutionary trajectories category* [142]. This is because, in the original study, the authors show that this measure correlates with the build latency and the number of best practice violations, which we directly compute (*i.e., Build time*, *Num. of docker smells*, and *Num. of shell script smells*). For the metrics of the category *Script Quality*, we use the *Hadolint* tool to detect violations of best practices. For the other metrics, we use a modified version of the parser from the replication package of the analysis conducted by Schermann *et al.* [105]. Specifically, we added the extraction of code comments, as their parser does not retrieve them. For the metric of the *Project Activity* feature, we use the tool

PyDriller to extract data from the source repository of each Dockerfile. For the *Build* category, we use the Python Docker wrapper[3] to build the Dockerfiles and measure their build time.

6.3.2 Experimental Procedure

This section details the experimental procedure we follow to answer our research questions.

RQ_1: Can the externally observable features explain the developers' preference for a Docker image?

To answer RQ_1, we extract the external metrics described in our taxonomy (Fig. 6.1) on the dataset D_{img}. We removed all the instances with invalid metrics values (*e.g., Clair scanner* fails on some Docker images), obtaining a total of 2,441 valid instances for the analysis. Next, to evaluate what are the external features that affect the developer preferences for a Docker image, we build two mixed-effect generalized linear models [33]. In detail, we use the *lmer* function from the R library *lmerTest*. Each instance of the dataset contains the value of the metrics for the external features, the application name and version, the number of adoptions, and the number of DockerHub stars. We use as random effects the application name and version. We use as random effects the application name and version. In this way, different Docker images regarding the same application at the same version, are considered in the same group. We do this because we want to take into account the fact that developers might have different levels of preferences for Docker images that provide different software applications, based on the characteristics of the applications themselves, regardless of the other image-related factors evaluated in our study. For example, the images `jdk-8-alpine` and `jdk-8-slim` will be in the same group, while `jdk-9-slim` and `jre-8-slim` will belong to other groups. The dependent variables, or outcomes, are the following:

- *Number of adoptions*: the actual usage in software repositories of a Docker image (*i.e.,* objective preference), measured as the occurrences of a specific

[3] `https://pypi.org/project/docker/`

Docker image (*i.e.,* name and tag) in user-defined Dockerfiles (as reported before);

- *Number of DockerHub stars*: the number of stars of a Docker image reported on DockerHub. This measures the prominence of a Docker image over others expressed by the developers. The number of adoptions and the number of stars tend to be directly proportional ($r_s = 0.23$, $p\text{-}value < 0.05$).

The independent variables (fixed effects in the model) are the following:

- *Image size*: the storage size of a Docker image, measured in bytes;
- *Num. of layers*: the total count of layers that compose a Docker image;
- *Num. of vulnerabilities*: the overall number of detected security vulnerabilities from a Docker image. All the vulnerabilities are considered (*i.e.,* from both parent and current image layers);
- *Image user is root*: whether the docker image uses the `root` account as the primary user;
- *Num. of exposed secrets*: total number of exposed secrets (*i.e.,* sensitive data) detected in the Docker image;
- *Is official*: the image is part of the Docker official images program, thus maintained following the official Docker guidelines.

All the independent variables refer to the external features described in Section 6.1.2. Before we performed the regression analysis, we applied some transformations to our dataset. First, we perform a correlation analysis to remove the highly correlated variables using a threshold of $r_s > 0.90$. None of the variables have been removed as their correlation coefficient remains below the threshold. Next, we computed the skewness coefficient of the distribution of all the variables. To normalize skewed distributions, we apply a logarithmic transformation to both dependent and independent variables (*i.e.,* $log(x + 1)$) since they are all non-negative. In our case, all the variable distributions are skewed (the lowest skewness value is 1.8, where a coefficient close to 0 means that the distribution is not skewed). Moreover, we apply a min-max normalization to fix the variables

on the same scale. As a result of our analysis, for each variable of our model, we report the significance value (*i.e., p-value*), the standard error, the coefficients, and the polarity of the relationship of that coefficients. We consider a coefficient *important* for determining the developers' preferences if it is statistically significant, *i.e., p-value* < 0.05. To evaluate the model fit, we report the adjusted R^2, using the *rsq* R package. It describes the variation explained by the model. Moreover, we report the effect size, expressed by measuring the Pearson correlation coefficient between pairs of independent and dependent variables [26] for the cases in which the relation, reported by the model, is statistically significant (*i.e., p-value* < 0.05). We also report Cohen's d effect size magnitude, obtained from Pearson's r by using the formula $d = \frac{2*r}{\sqrt{1-r^2}}$ [104].

RQ_2: Are configuration-related features correlated with externally observable features for Docker images?

To answer RQ_2, we compute the metrics related to the configuration features of our second dataset, *i.e.,* D_{src}. To perform the regression analysis on D_{src}, we built three mixed-effect generalized linear models. To explain how the external features are affected by the configuration features, we build a model for three of the external factors analyzed in RQ_1, as dependent variables, *i.e., Image size*, *Num. of vulnerabilities*, *Num. of exposed secrets*. We exclude from our regression modeling the external features *Is official* and *Image user is root* because the former is not an objective measure that depends on a set of non-quantifiable aspects, *i.e.,* is assigned by a team of Docker reviewers based on the official guidelines [19] and the latter can be directly controlled by the developer by adding a specific line of code. Also in this case, we consider the application name and version as a random effect. The independent variables (fixed effects in our models) are the metrics for the configuration features computed on the selected sample of Docker images (*i.e.,* D_{src}). In detail, the independent variables are the following:

- *Num. of docker smells*: number of best practice violations for Dockerfiles, extracted using the tool *hadolint*;
- *Num. of shell script smells*: number of best practice violations for shell script code used in Dockerfiles, extracted using the tool *hadolint*;

- *SLOC*: the total number of source lines of code (*i.e.,* without code comments and blank lines) in the Dockerfile;
- *Layer size*: the average number of commands executed in a single instruction block, to measure how much they are nested (*i.e.,* a proxy for the source code complexity);
- *Num. of docker instructions*: number of the used Docker instructions (*e.g.,* `RUN`, `FROM`, etc.) used in the Dockerfile;
- *Instructions entropy*: the Shannon entropy computed using the different Docker instructions used in the Dockerfile, as a measure for its complexity (*i.e.,* heterogeneity of the Dockerfile).;
- *Usage of additional script*: boolean flag that indicates whether or not the Dockerfile uses additional shell scripts, *i.e.,* it executes external scripts during the build of the Docker image;
- *Usage of external resources*: boolean flag that indicates whether or not the Dockerfile uses external resources, *i.e.,* it fetches additional data from remote sources (*i.e.,* URLs) during the build of the Docker image;
- *Usage of ENV*: boolean flag that indicates whether or not the Dockerfile uses environment variables, *i.e.,* identified by the instruction `ENV`;
- *Usage of build ARG*: boolean flag that indicates whether or not the Dockerfile uses build args, *i.e.,* identified by the instruction `ARG`;
- *Project age*: the age of the repositories that the Dockerfile belongs to, measured in seconds elapsed between the first and the last commit;
- *Num. of layers*: the number of layers that compose the Docker image, measured after the Dockerfile build;

Based on our hypotheses reported in Section 6.2, we define a model for each dependent variable, based on what we reasonably expect to impact each external feature. Specifically, for the outcome *Num. of exposed secrets*, we have as

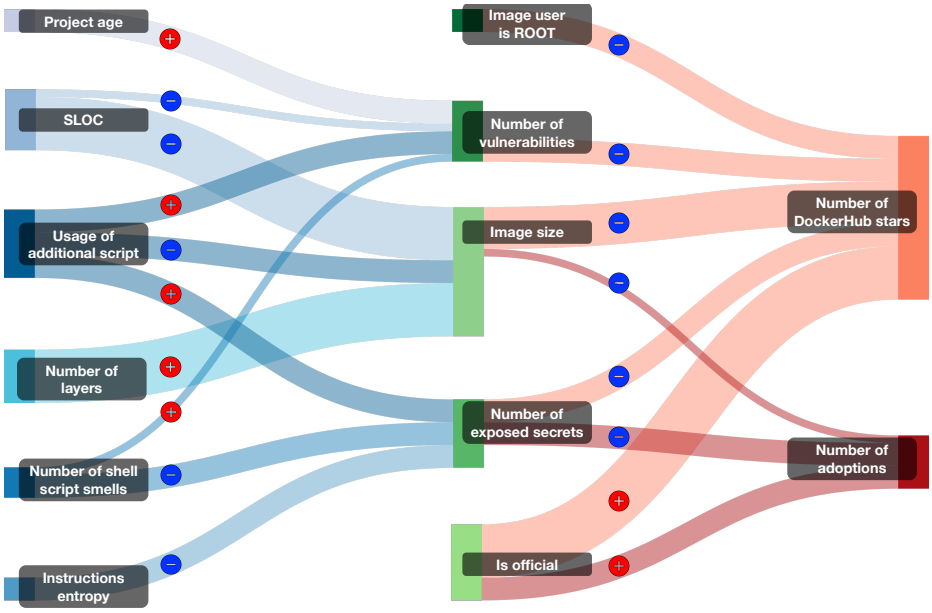
independent variables *Num. of docker smells*, *Num. of shell script smells*, *Instructions entropy*, *Usage of additional script*, *Usage of external resources*, *Usage of ENV*, and *Usage of build ARG*. For the outcome *Num. of vulnerabilities* we use as independent variables: *Num. of docker smells*, *Num. of shell script smells*, *SLOC*, *Usage of additional script*, *Usage of external resources*, *Usage of ENV*, *Project age*, and *Num. of layers*. Finally, for the outcome *Image size*, we have as independent variables: *SLOC*, *Num. of docker instructions*, *Layer size*, *Usage of additional script*, *Usage of external resources*, and *Num. of layers*.

We perform the same preprocessing steps done for answering RQ_1. First, we performed a correlation analysis to remove highly correlated variables (threshold of $r_s > 0.90$), but none were removed. Next, we evaluate the skewness coefficient. To normalize skewed distributions, we apply both square root and log transformations. In particular, we apply the log-transformation on the higher skewed distributions (skewness ≥ 1.8, *i.e.,* the metric *Num. of exposed secrets*), while the square root on the less skewed ones (skewness < 1.8). After this, we apply the min-max normalization to all of our variables. For each of our models, we compute the *p-value*, the standard error, the coefficients, and the polarity of the relationship of the coefficients with the dependent variable (*i.e.,* positive or negative). We consider a coefficient important for the dependent variable if the significance, *i.e., p-value*, is statistically significant (*p-value* < 0.05). As in the previous RQ, we compute the adjusted R^2 for each model, the effect size reported as Pearson's r between pairs of independent and dependent variables [26] and Cohen's d magnitude obtained from the correlation coefficient [104]. We do not report the results for all such models in the paper for readability reasons, but we discuss the main results, focusing on the relevant relationships we found. The detailed results are publicly available in our replication package [99].

6.3.3 Replication Package

Both the datasets (D_{img} and D_{src}) and the scripts we used to answer both our research questions are available in our replication package [99].

Figure 6.3: Descriptive plot of the relation between configuration-related features, externally observable features, and preferences for Docker applications. The size of the arrow indicates the effect size magnitude (*i.e.,* very small, small, medium, or large). The polarity of the relationship is reported with plus (positive) and minus (negative) signs.



6.4 Empirical Study Results

In this section, we report the results of our empirical study. Fig. 6.3 reports a summary of the relationships we found among configuration-related features and externally observable features, and then among external features and developers' preferences based on the results obtained from the two RQs. Connections indicate that the left-hand variable is significant in the model for explaining the right-hand variable. The size of the arrow represents the magnitude of the effect size (*i.e.,* very small, small, medium, or high). The polarity of the relation is reported through a plus (positive) or minus (negative) sign.

Table 6.4: Mixed-effects models obtained for explaining developers' preferences through external factors. The columns *Corr. Coeff* and *Effect Size* report the value of Pearson's r and Cohen's d magnitude, respectively.

	Variable	Estimate	*p-value*	Corr. Coeff.	Effect Size	Rel.
# adoptions	*Image size*	-0.0476	0.0274	-0.09	very small	↘
	Num. of vulnerabilities	-0.0047	0.6696		-	-
	Image user is root	0.0096	0.2644		-	-
	Num. of exposed secrets	-0.0538	0.0008	-0.07	very small	↘
	Is official	0.0904	< 0.0001	0.16	small	↗
# stars	*Image size*	-0.0937	0.0044	-0.26	medium	↘
	Num. of vulnerabilities	-0.0768	< 0.0001	-0.16	small	↘
	Image user is root	-0.0346	0.0104	0.12	small	↘
	Num. of exposed secrets	-0.1021	< 0.0001	-0.11	small	↘
	Is official	0.6014	< 0.0001	0.66	large	↗

6.4.1 RQ_1: Can the externally observable features explain the developers' preference for a Docker image?

We report in Table 6.4 the results of the performed regression modeling to explain the preferences for Docker images in terms of the number of adoptions and number of DockerHub stars, along with the Pearson's correlation between independent and dependent variables *Corr. Coeff*, and the effect size magnitude (*i.e.,* from Cohen's d). The variables *Num. of exposed secrets* and *Is official* are the most significant ones for the number of adoption, with a *p-value* < 0.001. That means developers tend to adopt official images, *i.e.,* images that follow the Docker official images program guidelines.

This is also true when considering the number of DockerHub stars as a dependent variable. This can be a consequence of the fact that they have few exposed secrets with a lower number of vulnerabilities (Fig. 6.4). The metric *Image user is root* is not statistically significant for the outcome *Number of adoptions*. This means that it does not influence the usage of a Docker image. On the other hand, it is significant for the outcome *Number of DockerHub stars* with a negative relation. This means that image users prefer images where the main account is not *root*. Fig. 6.4 shows the relation between each independent and dependent

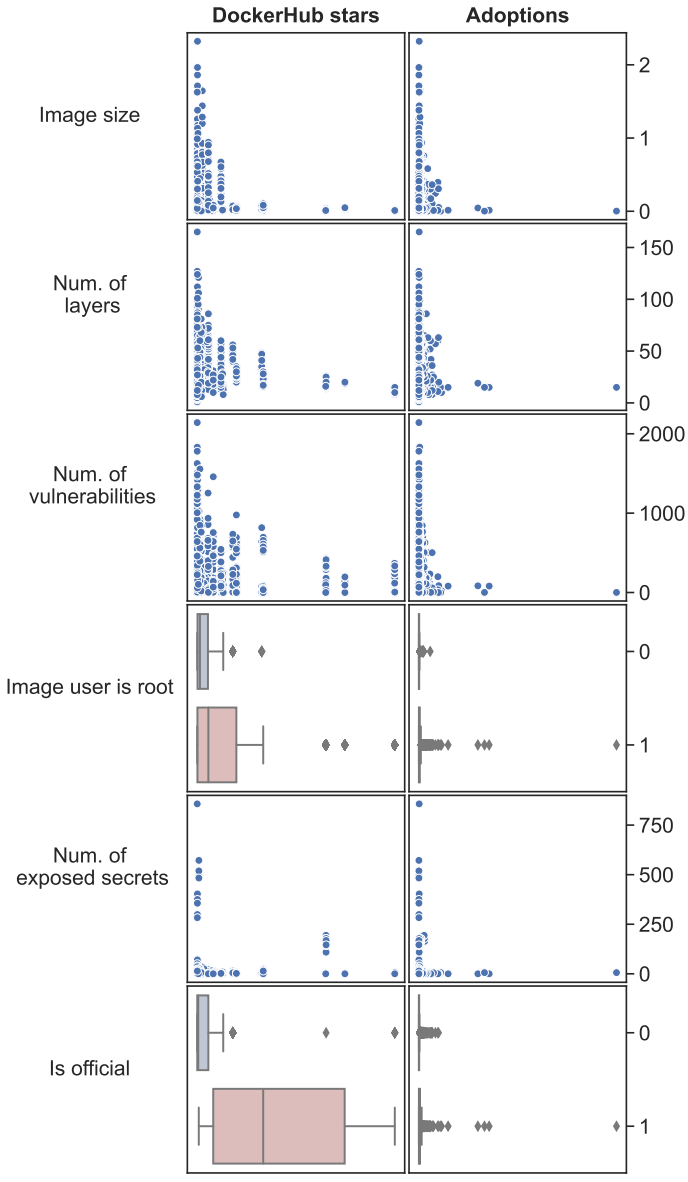


Figure 6.4: Descriptive plot of the relationship between dependent and independent variables for the regression modeling of RQ_1.

variable involved in RQ_1. We use boxplots for binary variables and scatter plots for continuous ones. We have an overall inverse relation between independent variables and outcomes, the higher the adoptions, the lower the external features of the Docker images. We computed the Spearman correlation between dependent and independent variables. The number of stars has a negative correlation with *Image size* and a positive one with the metric *Is official*. This means that the developers prefer smaller images having the official image label. A heatmap with the correlation values can be found in our replication package [99].

The adjusted R^2 for the two models are 0.18 (weak effect size) for the outcome *Number of adoptions*, and 0.74 (strong effect size) for the outcome *Number of DockerHub stars*. This shows that the external factors we considered are sufficient to explain the prominence of a Docker image over others expressed by developers. However, they are not enough to explain the actual adoptions. There could be other factors, still not investigated in the literature, that might help understand how developers choose the base images for their Dockerfiles.

Summary of RQ_1: The developers' preferences measured as *perceived* (prominence in terms of DockerHub stars) and *actual* (in terms of adoptions) can be explained by the image officiality-, security-, and size-related metrics. However, such metrics are much more effective in explaining the prominence, than the adoption.

6.4.2 RQ_2: Are configuration-related features correlated with externally observable features for Docker images?

We computed the Spearman correlation computed between configuration and external features of Docker applications. The highest correlation obtained is 0.75, between *Image size* with *Num. of layers*. When compared to *Layer size*, we have a negative correlation of -0.51. This means that large images have many layers that perform few actions, while in smaller images the number of layers is low and the number of actions performed is high. We also observe a negative correlation ($r_s = -0.28$) between *Usage of build ARG* and *Num. of*

exposed secrets: This is reasonable since developers might use build arguments to pass secrets (*e.g.,* passwords or keys) instead of having them hard-coded in the Dockerfiles themselves. A heatmap with the correlation values can be found in our replication package [99].

When combining such metrics in the three models we investigated, first, we found that the number of exposed secrets in the Docker image (*Num. of exposed secrets*) is higher when the Dockerfile uses additional scripts (*Usage of additional script*) and has a lower number of shell smells (*Num. of shell script smells*). The latter can be counter-intuitive. This is because if there are additional scripts, external to the Dockerfile, it is likely that the shell-script code is in them instead of inside the Dockerfile. Moreover, it is unlikely that shell-script smells can expose secrets in Docker images. The adjusted R^2 for such a model is 0.40 (weak effect size). We also observed that vulnerabilities (*Num. of vulnerabilities*) occur more frequently in older projects (*Project age*), when Dockerfiles are bigger (*SLOC*), they use additional scripts (*Usage of additional script*), and they have more shell-script smells (*Num. of shell script smells*). The adjusted R^2 for such a model is 0.24 (weak effect size). Finally, our results show that the image size highly depends on the number of layers (*Num. of layers*), as previously observed with the simple correlations. Similarly, the size is higher when the Dockerfile uses additional scripts (*Usage of additional script*) and fewer lines of code (*SLOC*). It is important to keep in mind that the size of a Docker image mainly depends on the number of layers and the base image used. For example, a Dockerfile that uses as base image the *nginx* web server, probably mainly performs the copy and the setup of the application to be contained. The adjusted R^2 is 0.76 (strong effect size). The detailed results of the models we built for RQ_2 are available in our replication package [99].

In summary, we observed that some configuration-related features have a significant role in explaining the external features we analyzed. In general, developers should keep the *SLOC* low to have benefits in terms of size and security. It is important to say that not all the lines of code (*i.e.,* instructions) have a direct impact on the image size (*e.g.,* the removal of non-functional instructions like `EXPOSE`). Similarly, developers should pay attention to the *Num. of layers*, which can negatively impact the size. Finally, the use of additional shell scripts

should be discouraged since it has a negative impact on both the security (*Num. of exposed secrets* and *Num. of vulnerabilities*) and size (*Image size*). Also in this case there are exceptions, *i.e.,* not all the shell script smells directly lead to security issues.

Summary of RQ_2: Some configuration-related features have a significant role in explaining the security and the size of Docker images. Developers should keep *SLOC* and *Num. of layers* low, and they should avoid using external shell scripts.

6.4.3 Discussion

From the results of our study, we can extract several hints that benefit both researchers and developers interested in improving the quality of their Docker images. The general picture is described in Fig. 6.3, which summarizes the outcome of the regression modeling for both RQs. We observed that Docker images having the highest number of adoptions have a small storage size and a low number of layers. Also, the number of exposed secrets is low, along with a low number of shell script smells, also avoiding the usage of additional scripts. The number of SLOC has to be low, along with the heterogeneity of instructions (*i.e.,* entropy).

The officiality of the image has resulted to be the strongest factor explaining the preference for Docker images, impacting both adoptions and stargazers count. For the latter, in addition to the features mentioned above, we have that image users prefer images with less number of vulnerabilities, where the main user of the image is not *root*. It is interesting to note that the number of vulnerabilities is positively affected by the repository age of the Dockerfile. This means, and confirms, that Dockerfiles must be actively maintained and updated to lower the presence of security vulnerabilities in the resulting images [109]. Also, the correlations found in our experiment are not strong for the specific metrics and features. Most likely, this happens because developers tend to pick official Docker images, with the assumption that they have the best quality overall.[4] We believe that this results from the fact that they are not aware of alternatives from the

[4] https://github.com/docker-library/official-images\#what-are-official-images

community of those images because it is difficult for users to compare similar Docker images as their peculiarities are not clearly highlighted [51]. An example is the `debezium/postgres:11` Docker image, where the source Dockerfile has fewer smells (*i.e.,* 6) compared to the official `postgres:11` (*i.e.,* 13). Another example is the `bitnami/nginx:1.19`, an unofficial Docker image for *nginx v1.12*, which has fewer security vulnerabilities (*i.e.,* 98) compared to the official image `nginx:1.19` (*i.e.,* 188). The behavior of the developers, when they pick a Docker image, could be related to the mismatch between adoptions and image preferences (*i.e.,* prominence), where we have a Pearson correlation $r = 0.25$ and medium effect size. We believe that, for the same reason, official Docker images tend to have more stars, *i.e.,* higher prominence ($r = 0.66$ and large effect size). We summarize some lessons learned from the results of our study.

Lesson 1. ***Image size* is influenced by *Num. of layers*.** Considering the results of our analysis, the number of LOC influences the number of layers. In Fig. 6.6 we report three different examples to qualitatively assess this relation. We have the Dockerfile *a* and a version with the number of layers reduced (*i.e.,* Dockerfile *b*) maintaining the same number of lines. Thus, if we build Dockerfile *a*, the resulting image will have 21 layers with a size of ∼315 MB. If we build Dockerfile *b*, the resulting image will have 15 layers with a size of ∼282 MB. In some cases, if a Dockerfile downloads an external package, the size of the resulting image will change independently of the number of layers and lines of code. For example, if we consider Dockerfile *b* with the two `RUN` instructions merged, compared to Dockerfile *c* where *tomcat* is installed using via *apt-get*, the resulting images will have the same number of layers, but the size of the former is higher than the latter (282 MB vs. 277 MB). Moreover, looking at Dockerfile *a* and *b*, it is clear that the number of layers is not related to the number of LOC but to the number of Docker instructions. However, we show an example where we modify instructions that directly impact the composition of the final image. The same does not apply to some kind of instructions, *i.e.,* removing instructions such as `LABEL` or `EXPOSE`. To the best of our knowledge, there are no automated tools for the refactoring of Dockerfiles that can help to reduce the image size. However, there is the *SlimToolkit*[5] that does not act on the Dockerfile,

[5] `https://github.com/slimtoolkit/slim`

but directly on the container. It creates a slimmed-down version of the Docker container maintaining the same functionalities.

Lesson 2. Shell scripts can be a proxy for security issues. An interesting point to discuss, resulting from our empirical study, is the fact that the usage of shell scripts can lead to security issues. There are mainly two types of shell scripts used in Dockerfiles: Embedded shell scripts and external shell scripts. For the former, the major issues are related to the best practice violations detected with the *hadolint*. For the latter, the main issue is that the shell script is executed in the same build context as the Docker image. In this way, it is possible to inject malicious code or access the host file system. In general, shell script code must be written in a safe way, following best practices, and additional scripts must be checked, or else they must come from trusted sources. It will be better to avoid copy-paste shell scripts from random websites. An example of a best practice violation that can expose the Docker container to security issues is the rule violation identified as `SC1098`, detected by the tool *hadolint*. The violation concerns the missing quote/escape for special characters when using the `eval` command. This rule is not a security issue itself, but its violation can lead to unpredictable outcomes from the script code. This can be exploited to inject malicious code.[6] Moreover, the main proxy for security vulnerabilities is related to the update status of the Docker images, *i.e.,* most updated images usually have fewer security vulnerabilities but are not exempt from them [109, 137].

Lesson 3. Dockerfile smells do not explain the adoption of the final Docker image. In the current scientific literature, the main measure to evaluate the quality of Docker images [13, 129] is the number of best practice violations (*i.e.,* smells) detected by the *hadolint* tool. Our results show that Dockerfile smells are not relevant for explaining any of the external factors we considered. In other words, their impact on the developers' preferences, when they have to choose whether they should adopt a Docker image, is negligible. Future work should be aimed at finding new types of smells, more related to the impact that they have on the resulting Docker image.

[6] `https://developer.apple.com/library/archive/documentation/OpenSource/Conceptual/ShellScripting/ShellScriptSecurity/ShellScriptSecurity.html`

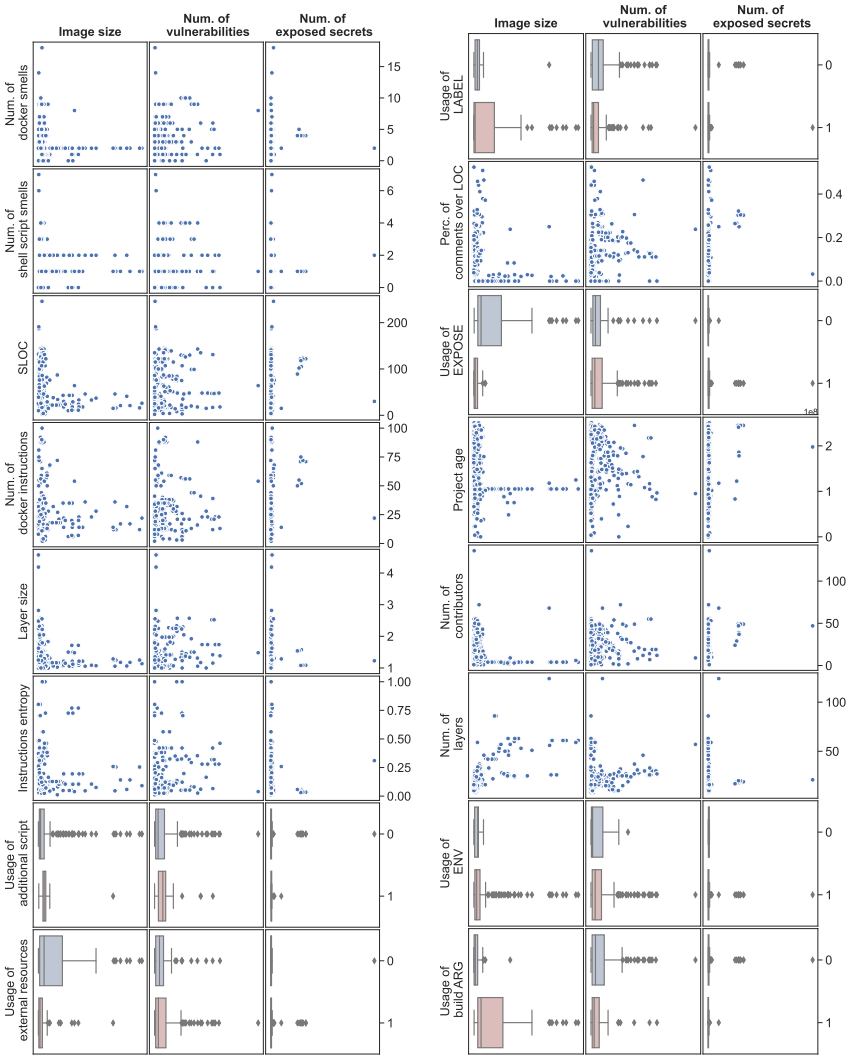


Figure 6.5: Descriptive plot of the relation between dependent and independent variables for the regression modeling conducted in RQ_2.

Figure 6.6: Examples of Dockerfiles having different image sizes.

```
1  FROM openjdk:7-slim
2
3  # INSTALL REQUIREMENTS
4  RUN apt-get update
5  RUN apt-get install --no-install-recommends -y wget
6  RUN apt-get clean
7  RUN rm -rf /var/lib/apt/lists/*
8
9  # INSTALL TOMCAT
10 RUN wget http://archive.apache.org/dist/tomcat/tomcat-7/v7.0.69/bin/apache-tomcat-7.0.69.tar.gz -O
       tomcat.tar.gz
11 RUN tar zxf tomcat.tar.gz
12 RUN rm tomcat.tar.gz
13 RUN mv apache-tomcat* tomcat
14
15 # ADD TOMCAT EXECUTABLE TO PATH
16 ENV PATH "$PATH:/tomcat/bin"
17
18 EXPOSE 8080
19
20 CMD ["catalina.sh", "run"]
```

(a)

```
1  FROM openjdk:7-slim
2
3  # INSTALL REQUIREMENTS
4  RUN apt-get update && \
5      apt-get install --no-install-recommends -y wget && \
6      apt-get clean && \
7      rm -rf /var/lib/apt/lists/*
8
9  # INSTALL TOMCAT
10 RUN wget http://archive.apache.org/dist/tomcat/tomcat-7/v7.0.69/bin/apache-tomcat-7.0.69.tar.gz -O
       tomcat.tar.gz && \
11     tar zxf tomcat.tar.gz && \
12     rm tomcat.tar.gz && \
13     mv apache-tomcat* tomcat
14
15 # ADD TOMCAT EXECUTABLE TO PATH
16 ENV PATH "$PATH:/tomcat/bin"
17
18 EXPOSE 8080
19
20 CMD ["catalina.sh", "run"]
```

(b)

```
1  FROM openjdk:7-slim
2
3  # INSTALL TOMCAT
4  RUN apt-get update && \
5      apt-get install --no-install-recommends -y wget tomcat7 && \
6      apt-get clean && \
7      rm -rf /var/lib/apt/lists/*
8
9  # ADD TOMCAT EXECUTABLE TO PATH
10 ENV PATH "$PATH:/usr/share/tomcat7/bin"
11
12 EXPOSE 8080
13
14 CMD ["catalina.sh", "run"]
```

(c)

6.5 Threats to validity

In this section, we report the threats to the validity of our study.

Construct Validity. We use state-of-the-practice tools such as *Clair* and *Hadolint*, to compute some of the metrics related to both external and configuration features (*e.g., Num. of vulnerabilities* and *Num. of docker smells*). To the best of our knowledge, the effectiveness of such tools for detecting the aspect that they aim at capturing was not validated in any previous study. However, such tools are already adopted both by developers in practice and researchers [109, 13, 129].

Internal Validity. To build our datasets we relied on the tool *GHSearch*, which provides all the software repositories from GitHub having more than 10 stars. While this could have biased the results towards more popular projects, we used this procedure to minimize the number of toy projects (*e.g.,* students' tests with Docker) in our datasets. While assigning the application name and version to each Docker image, we excluded the ones that contained more than an application name. We did this to avoid Docker images providing unique features that no other images could provide (*i.e.,* not comparable in terms of the environment alone). In doing so, we discarded 205 Docker images, which is negligible. An example of a discarded image is `tiangolo/uwsgi-nginx-flask:python3.5`[7]. It is worth saying that we only selected Docker images containing applications, so we discarded images for programming languages and OSs. Thus, we excluded a total of 128,704. Moreover, tagging some of the common programming languages and operating systems following the same procedure of Section 6.3.1, among the excluded images, we have 56,792 and 42,296, respectively. The remaining are uncategorized. In the first study, we ran a literature review to extract a collection of quality metrics that can impact the perceived quality of Docker images. We did not perform a Systematic Literature Review (SLR) on Docker quality to build the taxonomy because the topic is too broad, and it would have been outside the scope of the analysis conducted in this chapter. This is why we have not followed all the guidelines typically used to run a SLR [55]. As a result, we could have unintentionally excluded from our study some metrics defined in

[7] https://hub.docker.com/r/tiangolo/uwsgi-nginx-flask

the literature relevant four our study. However, we still tried to minimize this by using some of the guidelines defined by Kitchenham and Brereton [55]: First, we use precise inclusion and exclusion criteria (Table 6.1) to make sure we do not select irrelevant papers. Second, to enlarge the initial set of papers we selected, we both used snowballing (to include older relevant literature) and searched for papers that cite them (to include more recent literature).

External validity. Because of the procedure we used to build D_{img}, we started from Dockerfiles of downstream applications to define a list of Docker images to analyze. It is possible that, because of this process, we ignored some Docker images that are not used in open-source software but are used in proprietary software, such as *Oracle db*.[8] While it is clear that we could not have captured the number of adoptions for them without having access to a large amount of proprietary Dockerfiles, it is true that we could have done so for the number of stars, which is always publicly available. We decided not to have two different datasets for the two dependent variables used to answer RQ_1 to avoid obtaining incomparable results. For D_{src}, we manually looked for the Dockerfiles of a sample of Docker images provided for the top ten applications in terms of the absolute number of Docker images available. The results of RQ_2 might not generalize to all the applications we consider. Still, this procedure allowed us to cover about $\sim$50% of the total number of Docker image usages. It is important to clarify that our study was conducted on open-source Docker images and Dockerfiles, and, thus, our findings should not be generalized to other contexts (*e.g.,* industrial projects). In addition, the results come from a correlational study, where we cannot infer causality based on the data alone. In general, we reported practical examples to support our findings.

6.6 Final Remarks

There are plenty of Docker images available in public repositories, such as DockerHub, providing the same software systems and dependencies. It is unclear what aspects influence developers' preferences for an image over another, and also it is not clear what aspects describe the quality of a Docker image. In this

[8] https://hub.docker.com/_/oracle-database-enterprise-edition

chapter, we first presented a literature review of 31 papers to find what *externally observable features* and *configuration-related features* factors are typically considered. As a result, we defined a taxonomy of such features, along with the metrics typically used to measure them. Next, using such metrics, we performed an empirical study on a dataset of 2,441 Docker images to evaluate (i) what externally observable features impact the adoption of Docker images, and (ii) to what extent the configuration features influence external features. Our results show that the developers prefer Docker images that are official, secure, and small in storage size. Moreover, in terms of configuration features that have a significant impact on them, the *Num. of layers* must be kept low and *Usage of additional script* must be avoided if possible, where also the number of *Num. of shell script smells* must be low.

The takeaways of this chapter can be summarized as follows:

- The storage size of Docker images is influenced by the number of layers;
- Following best practices for writing Dockerfiles is not enough to explain the adoption of Docker images;
- However, following best writing practices for the shell script code contained in Dockerfiles can help to avoid security issues in images.

Based on the obtained results, future research could be aimed at defining a quantitative score for measuring the quality level of Docker images and Dockerfiles. Such a score could (i) help developers to choose among different alternative Docker images using a quantitative measure, and (ii) provide researchers with useful insights to build automated tools taking into account the quality level by objectively measuring it.

Acknowledgments

We thank the students Marco Russodivito and Stefano Fagnano (University of Molise, Italy) for the contribution given in the data extraction process.

CHAPTER 7

Towards the Automated Generation of Dockerfiles

Quality issues are not the only difficulty that developers have to face while writing Dockerfiles. In fact, the definition of a Dockerfile itself is not trivial: First, developers must have specific knowledge of the definition and setup of them (*e.g.,* define the execution environment); Second, while basic Dockerfile templates can be used as a starting point, they need to be adapted to the specific requirements of the software system at hand. To clarify this, let us consider the scenario in which a developer needs an environment that contains both `Apache Tomcat` as a web server and `FFMpeg` for processing videos, with the support of the x265 codec. While templates exist for the former, they do not contain hints about how to provide the latter. Fig. 7.1 shows an example of a possible solution: Such a Dockerfile starts from a pre-defined image which contains Tomcat, and it installs `FFMpeg` with the support of `x265` on top of it. It can be noticed that determining the commands required to achieve the latter goal requires a moderate effort, and it is prone to errors. For example, in the survey conducted by Reis *et al.* [96], the authors observed that developers (especially the less experienced ones) perceive the creation of Dockerfiles as a time-consuming activity. This problem has been

```
1  FROM tomcat:7.0.75-jre8
2
3  RUN echo deb http://archive.ubuntu.com/ubuntu precise universe multiverse >> /etc/apt/sources.list; apt
       -get update && \
4      apt-get -y --fix-missing install autoconf automake build-essential \
5      git mercurial cmake libass-dev libgpac-dev libtheora-dev libtool \
6      libvdpau-dev libvorbis-dev pkg-config texi2html zlib1g-dev \
7      libmp3lame-dev wget yasm && \
8      apt-get clean
9
10 WORKDIR /usr/local/src
11 # Install x265
12 RUN hg clone https://bitbucket.org/multicoreware/x265 && \
13     cd /usr/local/src/x265/build/linux && \
14     cmake -DCMAKE_INSTALL_PREFIX:PATH=/usr ../../source && \
15     make -j 8 && \
16     make install
17
18 WORKDIR /usr/local/src
19 # Install ffmpeg.
20 RUN git clone --depth 1 && \
21     cd ffmpeg && \
22     git://source.ffmpeg.org/ffmpeg && ./configure \
23     --extra-libs="-ldl" --enable-gpl --enable-libass \
24     --enable-libvorbis --enable-libx265 --enable-nonfree && \
25     make -j 8 && \
26     make install
27
28 WORKDIR /
```

Figure 7.1: Example of Dockerfile for `Tomcat` and `FFMpeg`.

observed in the literature and is related to the occurrence of Dockerfile smells (Section 2.2). Moreover, using Dockerfiles from tutorials and blob posts as a support for their creation leads, often, to broken Docker images [123].

As we reported in Section 2.4, various tools and approaches have been introduced to help developers write Dockerfiles. In summary, some of them [46, 82] takes as input the context and suggest whole Dockerfiles based on it. Such tools are handy since they allow developers to have a starting point quickly. They also require limited effort from the developers. However, they cannot recognize the need for specific libraries (like `FFMpeg`, in the previous example). Other tools provide more in-depth support but are limited to specific programming languages (*e.g.,* DockerizeMe for Python). The approach by Ye *et al.* [134] can recommend packages to install from an initial set of dependencies, but it only supports developers in defining the dependencies to install. The developer still needs to write complex instructions, like the ones required to build the `FFMpeg` and `x265` libraries Fig. 7.1. Finally, there are code completion tools that support developers while writing Dockerfiles [36]. Such tools, however, require that developers manually write part of the Dockerfile so that they can complete it, and none of them can generate complete Dockerfiles from a high-level description of what the developer

wants. Previous research [121, 79, 77, 126] show that DL is a viable solution for code generation-related tasks. However, to the best of our knowledge, no previous work tested to what extent DL can be used to generate complete Dockerfiles. We believe that more advanced support for writing Dockerfiles not only benefits developers, in terms of time and effort, but also the quality of the resulting Docker images. The resulting Dockerfiles can be less prone to errors and, in general, bad patterns.

In this chapter, we report a study aimed at filling this gap. We first propose a format for a structured high-level specification that serves to specify the requirements necessary in Dockerfiles. Then, we define a methodology for automatically inferring such a high-level specification (HLS) from existing Dockerfiles so that we can build a dataset large enough to train and test a DL model. To this aim, we rely on the largest collection of Dockerfiles available in the literature [27], containing 9.4M Dockerfile snapshots extracted from **all** the open-source projects hosted on GitHub. We run our specification-inference tool on them and, after a filtering procedure where duplicates and invalid Dockerfiles are removed, we end up with a set of 670,982 unique pairs ⟨HLS, Dockerfile⟩. We use this dataset to train and test a state-of-the-art DL model, the Text-to-Text Transfer Transformer (T5) [91], which has been proven effective when supporting several coding tasks [79, 77], following the same pipeline defined in the literature. We compare the DL-based approach with two Information Retrieval (IR)-based approaches, and we measure if the techniques are able to generate Dockerfiles that: (i) meet the input requirements, (ii) are similar to the target Dockerfile, and (iii) allow to build a Docker image similar to the target one. We obtain mixed results: While T5 achieves similar results to the best baseline in terms of adherence to the requirements, it generates Dockerfiles less similar to the target one. On the other hand, we found that the build of the Dockerfiles generated with T5 succeeds more often.

Interestingly, we also found that T5 truncates the Dockerfiles. This might be due to two main issues. First, **a larger training dataset might be needed for this task**: Despite we consider the largest collection of Dockerfiles in the literature [27], our results suggest that the T5 learning could benefit from more training data, considering the number of used instances to previous studies work-

Table 7.1: Format of HLSes, with the type of each field, a description, and the percentage of survey participants who indicated the field as important.

Field	Type	Description	% Positive Answers
OS	String	Operating system to be used.	75%
Package Manager	String	Linux package manager to be used (*e.g.*, `apt` or `apk`).	58%
Dependencies	String[]	Dependencies (*e.g.*, packages) that must be provided.	83%
Download of External Dependencies	Boolean	Dependencies can be installed as external resources (*i.e.*, not through the package manager).	58%
Usage of ENV	Boolean	Environment variables must be supported for customizing the container.	83%
Usage of ARG	Boolean	Arguments must be supported for customizing the image build process.	75%
Usage of LABEL	Boolean	Labels must be used for documenting the Dockerfile.	50%
Usage of EXPOSE	Boolean	Network ports used must be documented.	67%
Usage of CMD	Boolean	The command to execute when starting the container must be specified through the `CMD` or `ENTRYPOINT` command.	75%
Usage of ENTRYPOINT	Boolean		75%

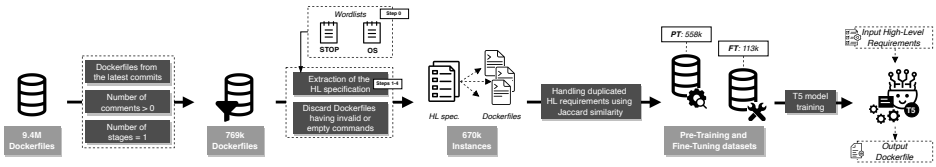


Figure 7.2: Steps performed to train T5 for generating Dockerfiles from specifications.

ing on source code [75]. Second, **a different T5 training stop criterion needs to be defined**: The stop criterion we adopt, which is the one currently used for coding tasks [77], is based on the convergence in terms of BLEU-4 score. However, considering our results, it proved to be ineffective for our context.

The remainder of this chapter is organized as follows. Section 7.1 we present the procedure to build the model for generating Dockerfiles from high-level specifications. In Section 7.2 we describe the design of our empirical study, while in Section 7.3 we report the results. Section 7.4 discusses the results and Section 7.5 the threats to validity of our study. Finally, Section 7.6 concludes the chapter by providing final remarks and open challenges.

7.1 Deep Learning for Generating Dockerfiles

We define a procedure to train a DL model for the generation of Dockerfiles from high-level specifications. In Fig. 7.2 we report the workflow we used to train a DL-based model (T5) for the generation of Dockerfiles. We first define

a structured high-level specification (HLS) for Dockerfiles. Then, we extract HLSes from existing Dockerfiles through an automated approach. Finally, we use such a dataset to train the T5 model for the generation of Dockerfiles (HLS → Dockerfile). In the following, we describe in detail the steps to construct our model.

7.1.1 Dockerfile High-Level Specification

Natural language can be used as an effective tool for reporting the requirements of the source code. When it comes to Dockerfiles, however, the high-level requirements that can be expressed are much more limited. For the source code, a developer might want to specify, for example, constraints on the input parameters and conditions that lead to errors. On the other hand, for Dockerfiles, it boils down to a matter of what the developer wants to be installed in the container, plus a few more characteristics.

Thus, to standardize the format of a Dockerfile requirements specification, written in natural language, the idea is to define a set of key-value requirements. In a real-world application, it could be seen as a structured form where, for each field, the developer specifies the values to meet the requirements. Based on the commands available in Dockerfiles and on how Dockerfiles are generally structured (based on our experience), we distilled a structured format for high-level specifications (HLSes), which we report in Table 7.1.

We did not include in the specification requirements related to particular Dockerfile commands (*i.e.,* `ADD`, `COPY`, `HEALTHCHECK`, `MAINTAINER`, `ONBUILD`, `SHELL`, `STOPSIGNAL`, `USER`, `VOLUME`, `WORKDIR`) for three reasons: First, they are related more to low-level details (*e.g.,* how the user of the Dockerfile should be set up); Second, in general, developers do not frequently use all of them [13, 27]; Third, some of them are deprecated (*i.e.,* `MAINTAINER`). Ideally, our approach automatically generates such low-level instructions when needed, *i.e.,* based on the instruction needed by the application that must be containerized. Also, developers might still easily tune them up on the generated Dockerfile, if they want to.

To validate the specification structure, we ran a survey in which we asked 12 professional software developers with at least 2 years of experience with Docker whether they would want to specify each of the fields we hypothesized to be

relevant in a HLS (binary yes/no question). The respondents come from personal invitation and social media, with a ∼40% response rate. We report the results in Table 7.1. All the fields are important to at least 50% of the participants. The least relevant field, according to the participants, is *Usage of LABEL* (50% positive answers), while the most important ones are *Dependencies* and *Usage of ENV* (83% positive answers).

7.1.2 Inferring Requirements from Existing Dockerfiles

We need a large amount of associations HLS → Dockerfile, to train a DL-based Dockerfile generation model. While Dockerfiles are largely available, this does not hold for the requirements behind them. The latter could be manually inferred, but such a process would be infeasible for a large-scale dataset. Thus, we defined an automated procedure for inferring the HLS behind an existing Dockerfile. In summary, the process works as follows. Given a list of operating systems (WL_{OS}) and stop words (WL_{stop}), we extract the operating system (step 1), the software dependencies (step 2), and, finally, the other fields required for HLS (step 3). In the following, we describe in detail each step of our methodology, and the procedure we used to define the two previously-mentioned lists of words.

Step 1: Inferring the OS

To infer the OS required by the developer, we only focus on the `FROM` instruction, which comes in the format `FROM <name>[:<tag>]`. Since `<name>` and `<tag>` are usually composed by one or more words, we extract them by splitting their content of image name and tag by the typically employed as separators, *i.e.,* `-` and `_`. We first check if any word in the WL_{OS} is present in the words extracted from the tag, and then in the name. If an OS-related keyword is present, we set *OS* with such a keyword, while we use the special keyword "any" otherwise, indicating that the developer did not have any requirement in terms of operating system. If more than an OS-related keyword is present, we consider the first match found. For example, for the instruction `FROM tomcat:9.0.20-jre8-alpine`, we extract the keywords (in the order): "9.0.20", "jre8", "alpine" (from the tag), and "tomcat" (from the image name). "alpine" is the first (and only, in this example) OS-related

word. Therefore, we set *OS* to "alpine". If the first OS-related keyword is found in the image name, we also add to it all the keywords containing only numbers from the tag name, which most likely refer to the version number. For example, for the instruction `FROM debian:10-slim`, we set *OS* to "debian10".

Step 2: Inferring Software Dependencies

Software dependencies might be present in several Dockerfile instructions. The first instruction which might contain dependencies is the `FROM` instruction. To detect the (possible) dependencies explicitly expressed in such a field, we extract all the words from the image name (with the same procedure described in step 1), we remove the words in $\mathrm{WL}_{\mathrm{OS}}$ (OSes) and $\mathrm{WL}_{\mathrm{stop}}$ (stop words), and we exclude the non-alphabetic words.

All the remaining words are added to the list of software dependencies (*Dependencies*). In the previous example, `FROM tomcat:9.0.20-jre8-alpine`, the dependency extracted (and only word used as image name) is "tomcat". At this point, we need to extract all the other dependencies installed with package managers or other procedures (*e.g.,* downloaded and installed). This task is far from trivial. Simply considering the packages installed through package managers (*e.g.,* `apt install`) is not an option: Most of the packages installed do not correspond to high-level requirements, but rather to low-level details about support libraries.

For example, in Fig. 7.1, the package `build-essential` is not a dependency of the software system, but rather a package incidentally needed (in this case, for building two actual dependencies, *i.e.,* `x265` and `ffmpeg`). We want our DL model to automatically infer the need of such packages.

However, it is known that developers tend to give an explanation comment of what each Dockerfile instruction does. Thus, we can use those comments to extract only high-level requirements, which are reported by the developers themselves. To achieve this, we use the following heuristic. We first select all the comment lines that contain the word "install". We tokenize each comment with the *spacy* Python library [28], and we detect the words that depend on such a keyword. We discard from the obtained list all the words in $\mathrm{WL}_{\mathrm{stop}}$, and we select the remaining ones as candidate dependencies for the specific comment line. For each comment line c_i with its candidate list of dependencies d_{c_i}, we process each

`RUN` instruction between c and the next comment line (c_{i+1}) or blank line. We use *bashlex*[1] for parsing the bash script in each RUN instruction, and we split it in statements. Finally, we consider only the statements in which the most common package managers (*i.e.,* `apt`, `yum`, `apk`, `pip`, and `npm`) and commands for downloading files (*i.e.,* `curl` and `wget`) are present, and we extract their arguments. If one of the arguments (packages) matches a candidate requirement (from the comment), we add the candidate requirement in *Dependencies*. Note that we do not simply consider the words depending on the "install" word from the comments because some words might not be software dependencies (*e.g.,* in "install only ruby", the word "only" should be ignored). Finally, we exclude duplicates, and we set the *Dependencies* with the dependencies extracted both from the `FROM` instruction and from the `RUN` instructions.

Step 3: Parsing Additional Fields

All the remaining fields are straightforward to be set. For the fields *Usage of ARG*, *Usage of CMD*, *Usage of ENTRYPOINT*, *Usage of ENV*, *Usage of EXPOSE*, and *Usage of LABEL*, we simply check if at least one of the respective instructions is present in the Dockerfile. The field *Package Manager* is defined by checking if any of the most commonly used package managers for Linux distributions (*i.e.,* `apt`, `apk`, and `yum`) are used in the `RUN` instructions. If this is the case, we set *Package Manager* to such a package manager (*e.g., Package Manager* = `yum`). We also check for the coherence between the package manager and the OS (Linux distribution) detected in step 1: For example, if the OS is `ubuntu`, the package manager can not be `yum`. If this happens, or if no specific package manager is detected (*e.g.,* no package is installed), we set *Package Manager* to `any`. Finally, for the field *Download of External Dependencies*, we check if any `RUN` instruction contains one of the following: (i) a link in the context of a download-related instruction (*e.g.,* `wget` or `git clone`); (ii) the installation of a Python or JavaScript external library; (iii) the installation of an external package through the package managers (*e.g.,* `dpkg` for Debian/Ubuntu). If this happens, we set *Download of External Dependencies* to `true` (`false` otherwise).

[1] `https://github.com/idank/bashlex/commit/58fe928`

Defining OSes and Stop Words

We use a systematic procedure for defining the two lists required by the HLS parser, *i.e.,* WL_{OS} and WL_{stop}. We extracted all the `FROM` instructions contained in the Dockerfiles from the collection by Eng *et al.* [27], that we later use to build our dataset. In total, we obtained 10,960,563 instructions. Then, we extract keywords from the image name and tag, like we do in step 1. Next, we discard the words that (i) contain only non-alphabetic characters (*e.g.,* version numbers), or (ii) have less than 3 characters (most likely stop words). We obtain a list of 46,070 unique words, along with the respective count of occurrences. We filter out all the words with less than 150 occurrences, thus obtaining 2,120 words, covering 97% of the total occurrences of all the words extracted. We manually analyzed and labeled each of them as "OS", "stop word", or "dependency". For example, "centos" is marked as "OS" keyword, "baseimage" as "stop word", while "python" as "dependency". In the end, we obtained 920 stop words for WL_{stop} and 74 words referring to OSs for WL_{OS}.

7.1.3 Defining a Dataset of HLSes and Dockerfiles

We proceed with using our parser to build the dataset of HLSes and target Dockerfiles. We use the dataset built by Eng *et al.* [27], which is the largest (9.4M) and the latest dataset of Dockerfiles currently available in the literature. That dataset comes from the S version of World of Code (WoC) [72], covering a period of time ranging between 2013 and 2020. To the best of our knowledge, this is the most recent and large collection of Dockerfiles available in the literature. The dataset provides all the versions (snapshots) of all the Dockerfiles extracted from GitHub projects, along with the related commit ID. For building our dataset, we only select the latest commit for each repository. As a result, we obtain 3,010,141 Dockerfiles.

We filter out all the Dockerfiles that have no comments (required in step 2 of the parser), and also those that have more than one stage, as they are currently not supported by our parser. Next, we drop all the duplicated Dockerfiles (according to their *sha1* hash) and all the Dockerfiles that contain in the `FROM` instruction at least a keyword that we did not manually evaluate when defining

the two lists $\mathrm{WL_{OS}}$ and $\mathrm{WL_{stop}}$, to ensure that we do not unintentionally include stop words as dependencies. After applying these filters, we obtain a total of 769,385 Dockerfiles, on which we run our approach to extract the HLSes.

At this stage, we further exclude all the Dockerfiles for which our parser detected syntax errors in bash instructions and empty Dockerfile instructions (*e.g.,* `COPY` without arguments). As a result, we obtain a total of 670,982 pairs of Dockerfile, associated with the respective HLSes (in total, we found 121,030 unique HLSes). At this stage, each HLS is associated with one or more Dockerfiles since different Dockerfiles might result from the same requirements. We need, however, to select a single representative Dockerfile for each HLS that we can use for the training and the test of T5. To do this, we first select all HLS associated with two or more Dockerfiles. We found 41,820 of them.

Given a set of Dockerfiles associated with the same HLS, we select the one containing the highest number of "typical" instructions for that cluster, to obtain the most typical Dockerfile for the given HLS. We tokenize each instruction of all the Dockerfiles. Then, given two Dockerfiles A and B, we compute the Jaccard similarity between each pair of instructions of the same kind (*e.g.,* `COPY` instructions are compared only with `COPY` instructions), with the formula $J(W_{A_i}, W_{B_j}) = \frac{|W_{A_i} \cap W_{B_j}|}{|W_{A_i} \cup W_{B_j}|}$, where W_{A_i} and W_{B_j} are the words from instructions i and j of A and B, respectively. We choose as representative the Dockerfile with the highest mean similarity over all the instructions.

We further process the Dockerfiles to prepare them for training: First, given all the package installation instructions (*e.g.,* `apt install`), we sort the packages in lexicographic order, to avoid that different package orders confuse the DL approach. Second, we remove all the lines that contain only comments, to avoid that the model makes extra efforts in solving a sub-problem which is not in the scope of our study.

From this, we extract two sub-datasets: $\mathrm{D_{PT}}$ for the pre-training, and $\mathrm{D_{FT}}$ for the fine-tuning of the T5 model. As for the former, we use all the Dockerfiles discarded while choosing the representative Dockerfiles for each HLS and all the Dockerfiles that are associated with a HLS for which the field *Dependencies* is empty (no software dependency needs to be installed). It is most likely that such Dockerfiles do have dependencies, but we were not able to find them because of

the lack of comments in the format we expected (*e.g.,* the word "install" has not been used in comments).

All the remaining pairs ⟨HLS, Dockerfile⟩ are placed in D_{FT}. In the end, we obtain 557,540 instances for D_{PT} and 113,442 instances for D_{FT}. As a requirement for the training of T5, we can only use Dockerfiles having no more than 1024 token, obtaining a total of 113,131 instances. Then, we divide the D_{FT} in training- ($D_{FT\text{-}train}$), evaluation- ($D_{FT\text{-}eval}$), and test- ($D_{FT\text{-}test}$) sets, by performing a typical 80%-10%-10% splitting [79, 77], obtaining 90,504, 11,313, and 11,314 instances, respectively. We use $D_{FT\text{-}train}$ for fine-tuning T5, $D_{FT\text{-}eval}$ for the hyperparameter tuning, and $D_{FT\text{-}test}$ for our experiment (see Section 7.2).

7.1.4 Training T5 for Generating Dockerfiles

Raffel *et al.* [91] introduced T5 to support multitask learning in Natural Language Processing. Such a model re-frames NLP tasks in a unified text-to-text format in which the input and output of all tasks are always text strings. A T5 model is trained in two phases: (i) *pre-training*, in which the model is trained with a self-supervised objective that allows defining a shared knowledge-base useful for a large class of tasks, and *fine-tuning*, which specializes the model on a downstream task (*e.g.,* language translation). T5 already showed its effectiveness in code-related tasks [79, 75, 78, 122, 62, 138, 11]. However, its application to the generation of Dockerfiles is novel and still unexplored. As done in previous work [79, 75], we use the smallest T5 version available (T5 small), which is composed of 60M parameters.

Given a prediction provided by the model, the output token streams can be generated using several decoding strategies. We use *greedy decoding* when generating an output sequence. In detail, such a strategy selects, at each time step t, the symbol having the highest probability of appearing in a specific position. We describe below both the pre-training and the fine-tuning procedure we applied for this task.

Pre-Training Procedure

The "general knowledge" [91] that we want to provide our model with is, in our case, a mixture of technical natural language (English) and technical language (Dockerfiles). We experiment with three pre-training variations: $T5_{NL}$, which only relies on natural language, $T5_{DF}$, which only relies on Dockerfiles, and $T5_{NL+DL}$, which relies on both. We test all such three variants, and we pick the best after performing hyperparameter tuning.

As for the first variant, $T5_{NL}$, we use the pre-trained checkpoint[2] released by Raffel *et al.* [91]. We do not perform any further pre-training for such a model.

Instead, we leverage the knowledge that has been already gained when pre-training the T5 model on the English text (C4 corpus [91]) for 1M steps.

As for the second variant, $T5_{DL}$, we adopt a classic *masked language model* task, *i.e.,* we randomly mask 15% of the tokens in a training instance, asking the model to predict them. We pre-train such a model on D_{PT}. Finally, as for the third variant, $T5_{NL+DF}$, we start from the $T5_{NL}$ model, and we further pre-train it for 500k steps on D_{PT}, using the same procedure used for pre-training $T5_{DF}$. Finally, we created a new *SentencePiece* model [59] for tokenizing natural language text. We trained it on D_{PT}. For both the models for which we performed additional pre-training steps ($T5_{DF}$ and $T5_{NL+DF}$), we used a 2x2 TPU topology (8 cores) from Google Colab with a batch size of 16 and a sequence length of 512 tokens for the input and 1,250 for the output. As a learning rate, we use the Inverse Square Root with the default configuration [91]. For the pre-training phase, we use the default parameters defined for the T5 model [91].

Hyperparameter Tuning

We test four learning rate strategies, *i.e.,* constant learning rate (C-LR), slanted triangular learning rate (ST-LR), inverse square learning rate (ISQ-LR), and polynomial learning rate (PD-LR). We report in Table 7.2 the parameters we use for each of them.

Given the three pre-trained models, $T5_{NL}$, $T5_{DF}$, and $T5_{NL+DF}$, we fine-tune them on $D_{FT\text{-}eval}$ (100k steps, batch size of 32, input sequence length of 512

[2] `gs://t5-data/pretrained_models/small`

Table 7.2: Configurations for the experimented learning rates

Strategy	**Parameters**	**Strategy**	**Parameters**
C-LR	$LR = 0.001$	ISQ-LR	$LR_s = 0.01$
ST-LR	$LR_s = 0.001$		$W = 10{,}000$
	$LR_{max} = 0.01$	PD-LR	$LR_s = 0.01$
	$Ratio = 32$		$LR_e = 0.001$
	$Cut = 0.1$		$Pow = 0.5$

tokens, output sequence length of 1024 tokens), leading to 12 different models (3 models $\times$ 4 strategies). Then, to assess their performance, we compute the BLEU-4 [86] metric between the generated Dockerfiles and the target ones. Such a metric has been used in previous work for other coding tasks [68, 61, 79, 77], and it ranges between 0 (completely different) and 1 (identical). We report in Table 7.3 the results achieved by the 12 models in terms of BLEU-4. The best results are achieved with $T5_{DF}$ with the ISQ-LR strategy and $T5_{NL+DF}$ with the ST-RL strategy (17.20% BLEU-4 for both).

In the end, we select the latter since $T5_{NL+DF}$ achieves better results also for the other strategies.

Fine-tuning

We fine-tune the best pre-trained model ($T5_{NL+DF}$) with the best learning rate strategy (ST-LR) on $D_{\text{FT-train}}$. We use early stopping to avoid overfitting [133, 122]: We save a checkpoint every 10k steps and compute the BLEU-4 score on the evaluation set every 100k steps. When the 100k steps do not lead to an improvement, we stop the training procedure, and we keep the last model.

Table 7.3: T5 hyper-parameter tuning results (BLEU-4).

Experiment	C-LR	ST-LR	ISQ-LR	PD-LR
$T5_{NL}$	13.80%	13.70%	5.50%	**14.50%**
$T5_{DF}$	16.90%	5.50%	**17.20%**	15.90%
$T5_{NL+DF}$	16.60%	**17.20%**	16.50%	17.10%

7.2 Empirical Study Design

The *goal* of our study is to understand to what extent T5 is effective in generating Dockerfiles. Our study is steered from the following research questions:

RQ_1: *To what extent is T5 able to generate Dockerfiles meeting the input natural language specification?* We evaluate the effectiveness of T5 in generating Dockerfiles that meet the input requirements.

RQ_2: *To what extent are the Dockerfiles generated by T5 similar to the original ones written by developers?* With this second research question we aim at understanding if T5 generates Dockerfiles similar to the targets ones.

RQ_3: *To what extent are the Docker Images built from the Dockerfiles generated by T5 similar to the original ones built form the Dockerfiles written by developers?* Two Docker images can be equal and come from completely different Dockerfiles. Therefore, with this last RQ, we try to understand whether the images built from the generated Dockerfiles are similar to the original ones.

7.2.1 Baseline Techniques

We use as baseline techniques two Information Retrieval (IR)-based approaches. The first one is IR_{ES}, and it is based on the state-of-the-practice for implementing IR approaches and search engines, *i.e.,* Elasticsearch [25]. Given a collection of documents (D) and a query (q), Elasticsearch first assigns a score to each document in D according to q, and then it sorts them. The score is computed with Okapi BM25 [24]. We add into an Elasticsearch instance all the instances

in $D_{\text{FT-train}}$. Specifically, for each instance, we define a document that contains both the HLS and the associated Dockerfile.

Given a new HLS for which we want to get a candidate Dockerfile, we perform a *boolean query* composed of all the fields of the HLS in OR clause (*i.e.,* `should`, in Elasticsearch). We report an example of the query in our replication package [98].

The second baseline we consider is IR_{ST}, and it is based on the *Sentence-Transformers* framework [95]. Such a framework allows to train embeddings for several data types (including text) so that they can be represented as numeric vectors. First, we use $D_{\text{FT-train}}$ to train the model for computing the embeddings (*bert-base-uncased*). Then, we compute the embeddings for each HLS as $e(d_{Spec})$ for each HLS in $D_{\text{FT-train}}$, and we store their associations with the respective Dockerfiles ($e(d_{Spec}) \rightarrow d_{Dockerfile}$). Given a new HLS, t_{Spec}, we compute its embeddings ($e(t_{Spec})$) and, then, the cosine similarity between $e(t_{Spec})$ and each $e(d_{Spec})$. Finally, we return the $d_{Dockerfile}$ for which the aforementioned similarity is maximum.

7.2.2 Context Selection

The *context* of our study is composed of (i) a set of associations HLS $\rightarrow$ Dockerfile, (ii) the Dockerfiles generated/retrieved by T5 and the two baseline techniques, and (iii) the source code of the software projects for which the Dockerfiles need to be built, for building the images and thus answering RQ_3.

As for the first object, we used the $D_{\text{FT-test}}$ dataset. To obtain the second object, we ran T5, IR_{ES}, and IR_{ST} on the HLSes from $D_{\text{FT-test}}$. As a result, we obtained three sets of generated/retrieved Dockerfiles, *i.e.,* DF_{T5}, DF_{ES}, DF_{ST}. Finally, to obtain the third object, for each Dockerfile in $D_{\text{FT-test}}$, we consider the original entry in the dataset by Eng *et al.* [27], and we recover the project from which it was extracted and the commit for that specific snapshot. We cloned all the repositories corresponding to the test instances, discarding the ones for which the source commit or repository was no longer available. As a result, we obtained a total of 3,909 repositories, corresponding to 4,059 instances of $D_{\text{FT-test}}$. Since building Dockerfiles requires a large amount of time, we did this for a representative sample of 500 Dockerfiles from $D_{\text{FT-test}}$ (4.28% margin of error, 95% confidence level). For each instance in $D_{\text{FT-test}}$, we tried to build

the original Dockerfile: If the build failed, we discarded the instance, while, if it succeeded, we kept it, until we collected 500 instances.

7.2.3 Experimental Procedure

To answer RQ_1, we compare the HLS related to the Dockerfiles returned by the approaches we consider with the one given as input. As for the two baselines, we already have an associated HLS for each returned Dockerfile (*i.e.,* the one from $D_{\text{FT-train}}$). This, however, is not true for T5 since it generates Dockerfiles from scratch. In this case, we use the same process described in Section 7.1.2 on the generated Dockerfiles for all the fields, except for *Dependencies*. We check if requirements in the input HLS are met in the generated Dockerfile. The reason is that we trained T5 not to generate comments.

Since our procedure for extracting *Dependencies* strongly relies on comments, we can not directly use it on the generated Dockerfiles. We ignore the Dockerfiles generated by T5 for which the parser we defined is not able to infer all the fields in the HLS (1,337 instances, *i.e.,* ∼12% of the total). We consider, instead, all of them for the two baselines, for which this cannot happen since, as mentioned, we do not infer the HLS. Finally, we measure the similarity between the target HLSes and the obtained ones. To achieve this, we assign a *score* for each field of the HLSes, compared to the respective instance in the target HLS, which is computed by assigning 1 point if the field is equal, and 0 otherwise. The only exception is the *Dependencies* field: Since this is a collection of elements, in this case we compute the score by computing the percentage of elements in the target HLS that are present also in the obtained one (*i.e.,* the recall). We compute and report the mean score obtained for each field of the HLSes.

To answer RQ_2, we compare the sets DF_{T5}, DF_{ES}, DF_{ST} with the respective target Dockerfiles. Simply computing textual similarity for Dockerfiles is not sufficient since, in many cases, instructions can be swapped without affecting the final result. Therefore, we rely on the AST representation of the Dockerfiles. To do this, we use the *binnacle* tool by Henkel *et al.* [44]. Binnacle extracts ASTs at three different abstraction levels. We use the *phase-2* representation. We do not use the *phase-3* abstraction level since it abstracts the bash commands. For example, both the `apt-get install` and `pacman -S` instructions get replaced with

a generic *package install*. While such an abstraction is useful, the tool does not support all the possible bash commands, thus causing loss of information, in some cases. We report an example of a parsed AST in our replication package [98]. Given the ASTs of two Dockerfiles, we compute the *edit distance* between them, *i.e.,* the number of modifications needed to transform one into the other, using the Zhang-Shasha algorithm [139]. We normalize the obtained edit distance by dividing it by the sum of the sizes of the two trees we are comparing. We discard all the instances for which the *binnacle* tool is not able to extract the AST. We obtain 9,963 for T5, 11,311 for IR_{ES}, and 11,129 for IR_{ST}. We run the Mann–Whitney U test [127] to compare T5 with the baselines in terms of normalized edit distance. The null hypothesis is that there is no difference between the edit distance obtained using T5 and the one obtained using an IR-based approach. We correct for multiple comparisons using the Benjamini-Hochberg procedure [7]. Finally, we compute the effect size using Cliff's Delta [14], which is negligible for all of the performed comparison (*i.e.,* T5 compared with the two baselines). This means that their difference is negligible, even if it is statistically significant.

To answer RQ_3, we compare the Docker images built from the resulting Dockerfiles provided by the three techniques and the one built from the target Dockerfile. We consider the GitHub projects we cloned for the sample of 3,909 instances and, for each of them, we replace the original Dockerfile with the one generated/retrieved by the three approaches, one at a time, and we try to build it. For each instance, we memorized (i) if the build succeeded, (ii) if the original and the obtained images are equal, and, if not, (iii) to what extent the latter provides what is also present in the former. As for the second measurement, we rely on the image digest: We say that two images are equal if their digest are equal. While this is not true by design, we can safely assume that, in our context, the risk of obtaining different images with the same digest is negligible. As for the third measurement, we compute the digest of each build layer (*i.e.,* one for each Dockerfile instruction), and we compute the percentage of layer digests of the image resulting from the original Dockerfile that also appear in the image built from the generated/retrieved Dockerfile. Note that if this measure is 100%, in this context, it means that the generated/retrieved Dockerfile is able to provide

Table 7.4: Adherence score between the input and the generated HLS reported for each field.

Approach	OS	Pkg Man.	Dep.	ENV	ARG	LABEL	EXPOSE	CMD	ENTRYPOINT	Down. Ext. Dep.
T5	**0.998**	0.981	0.865	**0.892**	**0.987**	**0.999**	0.798	0.743	0.843	0.816
IR_{ES}	0.922	**1.000**	**0.877**	0.812	0.884	0.872	**0.829**	**0.826**	**0.851**	**0.842**
IR_{ST}	0.880	**1.000**	0.761	0.518	0.168	0.165	0.373	0.453	0.260	0.448

everything that the original image already provided. Still, it is possible that it contains additional layers not present in the original image.

7.2.4 Data Availability

We provide all the data and scripts required to replicate our results in our replication package [98].

7.3 Empirical Study Results

In this section, we report the results of our study and, thus, the answers to our research questions.

7.3.1 RQ_1: Adherence to the High-Level Specification

In Table 7.4, we report the results of the comparison with input HLS fields. First, it can be noticed that IR_{ES} is the best-performing baseline, since it always achieves better or equal results than IR_{ST}. Therefore, from now on, we only discuss the comparison between T5 and such a baseline in this RQ. T5 is generally able to better meet the requirement in terms of *OS* (+7.6pp), while it achieves slightly worse results in terms of *Package Manager* (-1.9pp), *Dependencies* (-1.2pp), and *Download of External Dependencies* (-2.6pp). The last two are probably the most critical and hard-to-meet requirements since they also interact with each other, and we can observe that both the approaches generally achieve good results.

As for the other requirements, we observe that T5 performs better on *Usage of ENV*, *Usage of ARG*, and *Usage of LABEL*, while IR_{ES} achieves better results in terms of *Usage of EXPOSE*, *Usage of CMD*, and *Usage of ENTRYPOINT*.

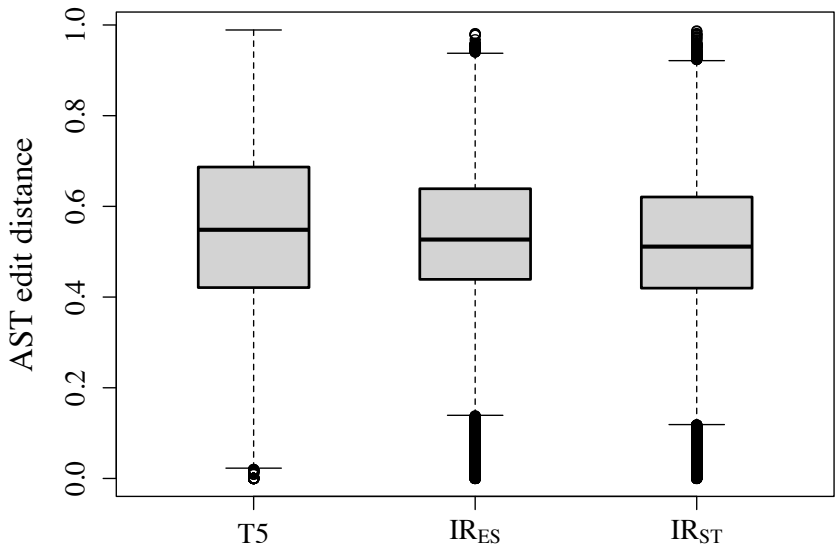


Figure 7.3: Boxplots of the normalized AST edit distance (RQ_2).

In summary, we can conclude that (i) there is no clear winner between the two approaches, and (ii) both the approaches generally return Dockerfiles that meet most of the requirements.

Summary of RQ_1: T5 and IR_{ES} perform very similarly in terms of adherence to the input requirements: There is no clear winner as for this aspect.

7.3.2 RQ_2: Dockerfile Similarity

We report the adjusted boxplots [49] for the normalized AST *edit distance* in Fig. 7.3. The higher the distance, the lower the similarity. Also in this case, there is no clear winner: T5 has higher variance, thus being able to generate both better and worse Dockerfile compared to the two baselines. The mean edit distance is 0.55 (σ = 0.19) for T5, 0.53 (σ = 0.18) for IR_{ES}, and 0.51 (σ = 0.19) for IR_{ST}. Therefore, the two IR-based baselines perform slightly better than T5. The difference is significant according to the Mann–Whitney U tests we performed for comparing T5 with the IR_{ES} and IR_{ST} (adjusted p-value lower

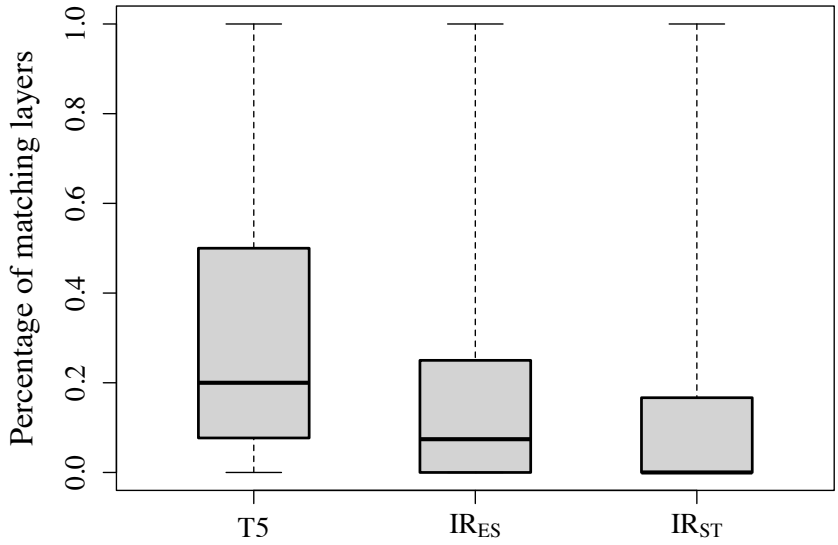


Figure 7.4: Boxplots of the percentage of matching layers (RQ_3).

than 0.001 for both). The Cliff's Delta between T5 and IR$_{ES}$ is 0.06, and 0.11 between T5 and IR$_{ST}$. Thus, the difference is *negligible* in both cases. We took a closer look at the cases in which the generated/retrieved Dockerfile was perfectly equal to the original one (*i.e.,* edit distance 0). We have 93 of such cases for T5, while only 18 and 11 for IR$_{ES}$ and IR$_{ST}$, respectively. In terms of AST size, the perfect matches are rather small (10.1, σ = 9.4, with a maximum of 58) compared to the average size (133.7, σ = 132.5) for T5, while it is remarkably higher for the two baselines (39.5, σ = 38.2 for IR$_{ES}$, and 41.8, σ = 44.8 for IR$_{ST}$). Such a result suggests that T5 works well when small Dockerfiles need to be generated, while it struggles with bigger ones. This is confirmed by the fact that the correlation (Spearman ρ) between the AST size and the edit distance is significant and high ($\rho = 0.67$), while it is not for the baselines (IR$_{ES}$: significant, with ρ = -0.12; IR$_{ST}$: not significant).

Summary of RQ_2: All the approaches returns Dockerfiles quite different from the target ones. T5 works well with small Dockerfiles, but not with bigger ones.

7.3.3 RQ_3: Docker Images Similarity

T5 achieves a build success rate of 34% (170/500 correctly built images), outperforming the IR_{ES} (23%, 166/500 images) and IR_{ST} (32%, 156/500 images). Comparing the *digest* of the built images (*i.e.,* hash value) with the source image (*i.e.,* the one built from the test instance), we obtain remarkably better results for T5: We have 11.7% of matches (20/170 instances), while the two baselines have no matching digest for their images. This result is confirmed also in Fig. 7.4, which depicts the distribution of the percentage of matching layers (adjusted boxplots [49]). The mean percentage of matching layers is 32.4% for T5 ($\sigma = 0.32$), 17.5% for IR_{ES} ($\sigma = 0.26$), and 10.9% for IR_{ST} ($\sigma = 0.19$).

The obtained results complement the ones presented in RQ_2. Not only T5 works better when it needs to generate small Dockerfiles, but it also works better than the two baselines on bigger ones, up to a certain point; after that, it is not able to generate good instructions, thus the limited layer match, in absolute terms.

Summary of RQ_3: T5 achieves the best results compared to the two baselines in terms of build success, percentage of perfectly matching images, and percentage of matching layers.

```
FROM golang:1.9.4-stretch

RUN apt-get update -y && apt-get upgrade -y

RUN git clone https://github.com/edenhill/librdkafka.git && \
  cd librdkafka && \
  ./configure --prefix /usr && \
  make && \
  make install && \
  cd ..

WORKDIR /go/src/consumerpg
COPY . .

RUN go-wrapper download
RUN go-wrapper install

CMD ["go-wrapper", "run"]
```

(a) Target Dockerfile

```
FROM golang:1.9.4-stretch

RUN apt-get update && apt-get install -y git

RUN git clone https://github.com/edenhill/librdkafka.git && cd librdkafka && ./configure && make &&
    make install

RUN go get github.com/confluentinc/confluent-kafka-go

WORKDIR /go/src/github.com/confluentinc/confluent-kafka-go

CMD ["go", "run", "main.go
```

(b) Dockerfile generated by T5

Figure 7.5: Example of a generated incomplete Dockerfile

7.4 Discussion

Generating Dockerfiles from high-level specifications is a challenging task. In this study, we perform a first attempt to solve this problem using Deep Learning (T5, specifically). Considering the overall results, there is no clear evidence that T5 is better than using IR-based techniques, in practice. Given the lower effort in setting-up an IR-based technique (which is trivial in the case of IR_{ES}, for example), at a superficial level, we can conclude that, at the moment, this would be the best option for practitioners.

However, we analyzed the results more in-depth to try to understand what went wrong, and why T5 does not work well for this task, despite it is works very well for other coding tasks [77]. First, we observed that T5 generates Dockerfiles with a much lower number of tokens compared to the two baseline approaches (36.80 vs. 135.70 for IR_{ES} and 107.10 for IR_{ST}). This explains why T5 works well for smaller Dockerfiles and gradually less well for bigger ones (RQ_2), and also why T5 achieves a good percentage of matching layers even if the Dockerfile similarity is low (RQ_3).

We manually analyzed some Dockerfiles generated with T5. We found that, in some instances, the Dockerfiles abruptly interrupt in the middle of the last instruction. While the remainder of the generated Dockerfiles is correct, the last instruction often contains issues. An example is provided in Fig. 7.5b, with the target Dockerfile (a) and the one generated by T5 (b). The prediction is remarkably good, until, in the last line, T5 stops the generation at a certain point. A pattern we observed is that interrupted Dockerfiles do not end with the token we used for indicating the new line (`<nl>`), while the ones in the training always end with such a token by design. We counted a total of 6,786 instances of such a kind (∼60% of the cases). In a real-world usage scenario, the developer would need to manually complete such Dockerfiles to make them work.

If we consider only the instances that terminate with the newline token, the results of T5 become better than the two IR baselines in all the aspects we considered: For RQ_1, T5 achieves better results than both the baselines for all the fields, except for *Package Manager*; for RQ_2, the edit distance is significantly lower, *i.e.,* 0.46, compared to 0.51 of the best baseline; For RQ_3, we obtain

results in line with the previously presented ones. We tried to address this issue by replacing the greedy decoding strategy with a sampling strategy, with different values for the *temperature* hyperparameter (T) of the *softmax* function. A high T increases the chances of picking tokens with low likelihood. When T is close to 0, this decoding strategy behaves like a greedy decoding strategy.

We tested temperature values between 0.7 and 1.0, with a step of 0.1. We observe that increasing the *temperature* allows to reduce the number of incomplete Dockerfiles: With a *temperature* of 0.7, we obtain a total of 1,648 incomplete generations (14.6%) which decrease to 1,283 (11.3%) with a *temperature* of 1. The average number of tokens contained in the Dockerfiles ranges between 82 (T = 1.0) and 84 (T = 0.7), while with greedy decoding we have 37, on average. In the end, however, the Dockerfiles generated with this strategy achieve generally worse results in terms of (i) number perfect predictions (∼-28%, with the best *temperature*, *i.e.,* 0.8), (ii) number of equal images (∼-80%, best T = 1.0) and layers (∼-20%, best T = 0.9).

This analysis shows that, while T5 learned how to generate Dockerfiles, to some extent, it does not have enough knowledge to generate complete Dockerfiles well: It either partially generates good parts of Dockerfiles or generates complete less-good Dockerfiles. There are two possible explanations for this phenomenon, which also represent open problems for this specific task:

- **A larger dataset needs to be built.** Addressing this problem is only apparently easy. We considered in our study the largest collection of Dockerfiles available in the literature [27], which includes *all* the Dockerfiles from open-source projects produced up to 2020. While such a dataset can be updated with the last two years of activities in GitHub, it is unlikely that the size of our dataset would drastically increase as a result. Indeed, we consider a single Dockerfile for each unique HLS, *i.e.,* we would not have new instances for the HLSes already covered. We rely on comments by the authors to extract some requirements (specifically, the *Dependencies* field). Some Dockerfiles, however, do not have explicit indications of such requirements and, therefore, we miss them in the inferred specifications. New and more precise ways of extracting high-level requirements from Dockerfiles are needed. Also, a promising direction would consist in the definition of

techniques for data augmentation in this context, *e.g.,* by blending existing Dockerfiles to provide uncovered combinations of HLSes.

- **A different training stop criterion needs to be defined.** In this study, we used the same procedure previously used for coding tasks [77], for which the problem of abrupt interruption in the inference has not been observed. It is possible that the stop criterion and the metric (*BLEU*) used are not the right ones in this context. As for the latter, it might be worth exploring different distance measures. While the AST distance is not a viable option for performance reasons, other metrics that do not consider the order in the instructions might be more useful.

As a takeaway for practitioners, it may be too early to reliably use these approaches to generate entire Dockerfiles in one shot, without the need to make adjustments.

7.5 Threats to Validity

Threats to Construct Validity concern the correct operationalization of the concepts being studied. First, the inference method we used for extracting requirements from Dockerfiles works on a series of assumptions (*e.g.,* the presence of comments) that might not always be completely satisfied in practice. To mitigate this crucial threat, we carefully tested our parser and manually checked examples until we were satisfied with the procedure used to achieve this goal. In total, we were able to infer 113,442 unique HLSes, with 31,990 unique combination of dependencies, which gives us confidence on the fact that our parser works as intended for most of the Dockerfiles considered. Also, we made sure to exploit all the instances (also the ones without comments) in the training procedure (*i.e.,* by using them for pre-training). The choice of the fields that compose our HLS could exclude requirements that developers might be interested in specifying (such as `USER`). As we explained in Section 7.1, we exclude only the requirements derived from Dockerfile instructions that do not appear very often. The best candidate Dockerfile we selected for each HLS (Section 7.1) might not be representative of

the group of Dockerfiles. In our methodology, we relied on Jaccard similarity to mitigate this risk.

Threats to Internal Validity concern factors internal to our study that might have influenced our findings. The percentage of marching layers (RQ_3) might not accurately capture the structural similarity between two Docker images: If a layer is different, many or even all the subsequent layers will be different. However, to the best of our knowledge, the only alternative is to perform a diff on Docker containers [37], which, however, only works at the level of installed packages. Considering the layers allows knowing from which point the two images started to differ, we believe this is the best choice.

Threats to External Validity concern the generalizability of our findings. We relied on the largest collection of Dockerfiles available in the literature [27]. More Dockerfiles might have been created between 2020 and now. However, we believe that the considered dataset allows us to provide reasonably generalizable.

7.6 Final Remarks

Writing Dockerfiles is a time-consuming and error-prone activity for developers. Since Deep Learning has been successively applied to solve code-related tasks, we evaluated its effectiveness (and, specifically, T5) for the automatic generation of Dockerfiles. The aim is to support developers while writing Dockerfiles to reduce the effort and the risk of introducing quality issues. To achieve this, we trained a T5 model on the largest dataset available of open-source Dockerfiles, from the literature. We compared the model with two IR-based baselines, *i.e.,* IR_{ES} and IR_{ST}. The results show that, while T5 works very well on small Dockerfiles, it struggles with larger ones.

We can summarize some lessons learned as follows:

- DL-based approaches are not ready for production yet. While the results are promising, they are not good enough to be used in practice. In particular, T5 is able to generate complete Dockerfiles in many cases, thus requiring manual intervention;

- Information Retrieval-based approaches are a viable alternative, which is also easier to set up and use. In particular, IR_{ES} is the best-performing baseline in our study;
- A large dataset is needed to train a DL-based approach for this task. The dataset we used is the largest available in the literature, and it is not easy to build a larger one since the current one has been built by considering all the Dockerfiles from open-source projects up to 2020. New techniques for data augmentation could help to address this issue, or else reduce the complexity of the task by using abstractions for the non-functional instructions (*e.g.,* `ENV` or `LABEL`).

After having deeply analyzed this phenomenon, we identified two possible issues that must be addressed to build an effective DL-based solution for this task. In summary, the open challenges to face are:

- First, it is necessary to *build a larger dataset*, which is not easy, given that we used the largest collection of open-source Dockerfiles available;
- Second, it is necessary to *devise a different stopping criterion* for fine-tuning T5 since the one typically used for coding tasks (based on BLEU-4) likely causes an early stop, which does not allow the model to properly complete the learning process.

Other interesting points to investigate as a future research agenda are (i) the generation of *docker-compose* files, which are commonly used in Docker-based projects, and (ii) making context-aware generations, in terms of build and execution environment.

CHAPTER 8

Conclusion

Ensuring the delivery of high-quality Docker artifacts is crucial for a successful software application. However, the occurrence of design flaws and quality issues, known as *smells*, may hinder the quality of Dockerfiles first, and of Docker images then. This could lead to several problems, such as security vulnerabilities, performance issues, and maintainability problems. This chapter concludes the thesis by summarizing the contributions made and by discussing the lessons learned. It also outlines and discusses several future research directions.

The existing literature presents some limitations and gaps that we tried to address in this thesis. In particular, (i) the current literature investigated the diffusion of smells in Dockerfiles, along with the definition of new catalogs and approaches to detect them, but none of the previous research investigated smells from the point of view of developers; (ii) the current literature lacks tools to support developers to improve their Dockerfiles, by providing automated fixes for the most common best practices violations; (iii) even if several studies have been presented in the direction of quality features and improvement of Docker images, it is still not clear what are the most important quality aspects that

characterize Docker images and how to improve them; (iv) there is a general lack of advanced tools to support developers in the development and maintenance of Docker artifacts, which could help to reduce the effort and the quality issues introduced when developing Dockerfiles. The research goal of this thesis aims to address those limitations. In particular:

> **Goal**: *This thesis aims to advance the state-of-the-art of development and maintenance of quality-aware Docker artifacts by providing developers and researchers with tools and insights related to what quality features and issues characterize them.*

From this, three different research obectives were defined. In summary:

- **RO1**: *How do developers perceive Dockerfile smells?*
- **RO2**: *What quality aspects developers prioritize in Docker images?*
- **RO3**: *How can we support developers to define better Dockerfiles in terms of quality?*

To address the first research objective (RO1), we conducted a survey involving 37 expert developers in Docker development to assess the relevance of the 17 most diffused smells included in the *hadolint* catalog (Chapter 3). The results showed that not all the smells are considered a problem to address by developers, and they prioritize the writing practices related to the performance more than the code quality itself. Moreover, we investigated how developers address Dockerfile smells in the open-source (Chapter 4). In detail, we manually investigated a sample of 1,000 smell fixing commits and evaluated an automated fixing tool for 12 of these smells via pull requests. The results showed that developers are aware of smells, and are willing to address them. In particular, smell impacting the performance are those more preferred.

The results achieved so far lead us to the second research objective (RO2). In particular, we investigated what changes developers apply to improve the performance of Docker images (Chapter 5), specifically by looking at the operations performed on Dockerfiles to reduce the build time and image size. We propose a taxonomy of 54 performance-related changes, extracted from 383 commits, and

Table 8.1: A summary table of the open data and code related to the contributions made in this thesis.

RQ	Chap.	Description	Ref.
1	3	We provide the data of the smell occurrence in the official Docker repositories and the tasks proposed to the participants of the survey	[2]
	4	We provide the raw commits data and the validated smell fixing commits. Also, we provide the code of DockLeaner, along with the raw results of the validation	[102]
2	5	We provide the raw commit data and the NLP-based tool used to filter them. We also provide the raw results of the build and size measurement	[3]
	6	We provide the data of the evaluated Docker images and their source Dockerfiles, along with the tools to extract the quality metrics. We also provide the code to build and execute the involved statistical models	[99]
3	7	We provide the train, validation and test data. We also provide the code to train the T5 model, the resulting model, the code for the two baselines, and the tool to parse the high-level requirements	[98]

we measured their impact on the build time and image size (Chapter 5). The obtained results guide developers and researchers on what are the most effective changes to apply to improve the performance of Docker images.

Additionally, we conducted a literature review to propose a taxonomy of the quality features and metrics characterizing Docker images (Chapter 6). We evaluated their relationship with the adoption and prominence of a Docker image over others. The results showed that developers tend to choose images that are smaller, secure, and part of the *official images program*.

Finally, we addressed the third research objective (RO3) by empirically evaluating the applicability of deep learning for the generation of Dockerfiles. We compared the state-of-the-art T5-base model with two information retrieval baselines (Chapter 7). The results showed that the T5-base model outperforms, even if slightly, the two baselines. However, the approach is not ready-to-use due to some issues that are still present in the generated Dockerfiles. Thus, we conclude that the approach is promising, but it needs further investigation to be used in practice.

The research contribution of this thesis is not limited only on the obtained findings. We made publicly available the data and the code used in the studies reported in the previous chapters to allow reproducibility and contribute to future research directions. We summarize the open data related to the contributions made in this thesis in Table 8.1.

8.1 Lessons Learned

The results obtained from the studies conducted in this thesis led to several lessons learned and considerations. We summarize them in the following:

- **Lesson 1. Dockerfile smells are not always considered a problem by developers.** The results of the survey conducted in Chapter 3 showed that developers are not always aware of the smells that may occur in Dockerfiles, and they do not consider all of them as a problem. This is related to the fact that not all the existing smells have a concrete impact on the final Docker image, but are only a style issue. An example is the smell DL3003 (Use `WORKDIR` to change directory). The smell has not been considered an issue since there is no observable difference in following the practice. Even, it could increase the size of the final image since it adds an additional layer. Nevertheless, the missing version pinning of the base image and the package dependencies (*e.g.,* installed via `npm` or `pip`) is considered a serious issue since it can lead to various problems impacting the reliability of the final image.

- **Lesson 2. Developers are more interested in performance than code quality.** Even if we spent a lot of effort trying to understand the point of view of developers, the solution turned out to be the simplest: Performance is the key. The results from Chapter 3 and Chapter 4 showed that Dockerfile developers give a lot of attention to the performance of their code. Due to the nature of the container, the developers that work on Dockerfiles might have more of an *operations* background than a software development one, thus prioritizing the technical aspects of the artifacts they produce. Thus, the current catalogs of smell are not comprehensive enough to cover all the possible issues that may occur in Dockerfiles.

- **Lesson 3. The context in which a smell appears is crucial.** *Hadolint* is the most popular tool to detect Dockerfiles smells, however, it is not perfect. In Chapter 4 we found out that in some cases the detected smells are false positives, mainly because the way in which they happen is not in line with the writing practice that the rule violation captures. For example,

rule DL3007 (Use a specific tag instead of `latest`) is triggered when the `latest` tag is used. In some cases, developers use that version tag because they want the most up-to-date base image. The same is for the previously mentioned DL3003: Using the pattern `cd <path>` in a `RUN` instruction, to temporarily clone a repository, is not a smell, but a common practice. However, these exceptions are not true for all the smells. For example, the smell DL3006 (Always tag the version of an image explicitly) is always considered a problem, since it may lead to several problems, such as the impossibility of reproducing the same image in the future.

★ **Lesson 4. Base images are the key.** The results of Chapter 6 showed that the base image is the most important aspect that developers consider when selecting a Docker image. This is related to the fact that the base image is the starting point of the definition of a Dockerfile. Thus, starting from a bad base image leads to a bad Docker image, simply. Moreover, the results showed us that developers prefer to use official images since they are by design more reliable and secure. Besides, choosing more efficient base images is the most impacting factor in reducing the storage size and build time (Chapter 5).

★ **Lesson 5. A lot can be done towards the automated generation of Dockerfiles.** In Chapter 7 we investigated the applicability of the current state-of-the-art deep learning technique in generating Dockerfiles. We had to address several issues, such as the lack of a large training data, and several modifications to the training procedure, *i.e.,* experimenting with different hyperparameters. Also, since a lot of Dockerfiles inherit a part of the configuration from the base image, there is not a clear number of instructions required to meet a specific requirement. Our work was conducted before the rise of *ChatGPT*, which could be a good candidate able to address the issues we faced. However, a lesson that we can learn from that tool is that an iterative generation approach, instead of generating a Dockerfile in one shot, could help to obtain an overall better solution at the end.

8.2 Future Research Directions

The research in the domain of Docker quality is still in its infancy, where the first study about the occurrence of smells was conducted in 2017. The Docker platform is relatively new since it was introduced only in 2013 but widely adopted in the last few years. The Docker platform is becoming more and more popular and embraced by the industry, and thus the need for more advanced tools and techniques to support the development and maintenance will be more and more important. In this section, we outline some future research directions that could be investigated as future research contributions.

- **Future Work 1. Validation of the smell catalogs and detection approaches.** While it is true that the current catalogs have been proposed starting from the suggestions of developers, or resulting from empirical investigations, we found that some of them are not identified as smells, Also, the existing tools to detect smells are not perfect, and they may lead to false positives. Future studies should focus on (i) validating the existing catalogs by interviewing developers, and (ii) empirically evaluating the current state-of-the-practice in smell detection (*i.e., hadolint*) to improve both the tool and understand the implications of the false positives in studies involving it.

- **Future Work 2. A systematic large-scale review of Dockerfile smells.** The current literature proposed different catalog of smells (*e.g., Binnacle* [44] and *DRIVE* [146]). What we proved in this thesis is that the current catalogs have some missing pieces regarding the information provided (*i.e.,* high-priority smells should be separated from low-priority ones), and the captured writing practices (*i.e.,* we need more smells related to build time, image size, and security). However, the *gray literature* offers a lot of different insights and alternative suggestions on how to write good Dockerfiles.[1] Thus, a future direction could be to conduct a large-scale review of Dockerfile smells, by involving a large number of developers and also including the information from gray literature, to define and validate

[1] `https://github.com/hexops/dockerfile`

a more comprehensive catalog in which the writing practices converge to a common set of rules.

- **Future Work 3. More and more advanced tools to support developers are needed.** The contribution of this thesis advanced also the state-of-the-practice by providing DOCKLEANER, and comparing different tools to automatically define Dockerfiles (*e.g.,* T5 and IR) reducing the effort required by developers to write them. Future works can also contribute in that direction, by improving the existing tools where it is known that they have limitations (*e.g.,* the false positives caught by *hadolint*). Other suggestions that arise from the results of this thesis are: (i) Improve the effectiveness of *hadolint*, (ii) extend the smells supported by DOCKLEANER (it will be released as an open-source tool), (iii) propose approaches aimed at measuring the quality level of a Docker image (based on the insight of Chapter 6), and also (iv) define automated tools able to optimize the performance of Docker images (*e.g.,* by leveraging the taxonomy proposed in Chapter 5).

Appendices

APPENDIX A

Publications

J1 **Rosa, G.**, Scalabrino, S., Bavota, G., & Oliveto, R. (2023). *What Quality Aspects Influence the Adoption of Docker Images?*. In ACM Transactions on Software Engineering and Methodology, 32(6), 1-30.

P1 **Rosa, G.**, Scalabrino, S., & Oliveto, R. (2022). *Fixing Dockerfile Smells: An Empirical Study*. In 38th IEEE International Conference on Software Maintenance and Evolution (ICSME), Registered Reports Track. arXiv preprint arXiv:2208.09097.

C3 **Rosa, G.**, Scalabrino, S., Robles, G., & Oliveto, R. (2024). *Not all Dockerfile Smells are the Same: An Empirical Evaluation of Writing Practices by Experts*. In IEEE/ACM 21th International Conference on Mining Software Repositories (MSR) (pp. To Appear). ACM.

C2 **Rosa, G.**, Mastropaolo, A., Scalabrino, S., Bavota, G., & Oliveto, R. (2023). *Automatically Generating Dockerfiles via Deep Learning: Challenges and Promises*. In 2023 IEEE/ACM International Conference on Software and System Processes (ICSSP) (pp. 1-12). IEEE.

C1 **Rosa, G.**, Scalabrino, S., & Oliveto, R. (2022, October). *Assessing and Improving the Quality of Docker Artifacts*. In 2022 IEEE International Conference on

Software Maintenance and Evolution (ICSME), Doctoral Symposium Track (pp. 592-596). IEEE.

A.1 Other Publications

J6 **Rosa, G.**, Russodivito, M., Laudato, G., Colavita, A., De Vito, L., Picariello, F., Scalabrino, S., Tudosa, I., & Oliveto, R. (2023). *ST-Segment Anomalies Detection from Compressed Sensing Based ECG Data by Means of Machine Learning.* In International Joint Conference on Biomedical Engineering Systems and Technologies(pp. 237–255), Springer Selection.

J5 **Rosa, G.**, Pascarella, L., Scalabrino, S., Tufano, R., Bavota, G., Lanza, M., & Oliveto, R. (2023). *A comprehensive evaluation of SZZ Variants through a developer-informed oracle.* Journal of Systems and Software, 202, 111729.

J4 Piantadosi, V., **Rosa, G.**, Placella, D., Scalabrino, S., & Oliveto, R. (2023). *Detecting functional and security-related issues in smart contracts: A systematic literature review.* Software: Practice and Experience, 53(2), 465-495.

J3 **Rosa, G.**, Russodivito, M., Laudato, G., Colavita, A., Vito, L., Picariello, F., Scalabrino, S., Tudosa, I., & Oliveto, R. (2021). *Multi-class Detection of Arrhythmia Conditions Through the Combination of Compressed Sensing and Machine Learning.* In International Joint Conference on Biomedical Engineering Systems and Technologies (pp. 213–235), Springer Selection.

J2 **Rosa, G.**, & Pareschi, R. (2021). *Tether: A Study on Bubble-Networks.* Frontiers in Blockchain, 34.

C6 Guglielmi, E., **Rosa, G.**, Scalabrino, S., Bavota, G., & Oliveto, R. (2022, October). *Sorry, I don't Understand: Improving Voice User Interface Testing.* In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (pp. 1-12).

C5 **Rosa, G.**, Russodivito, M., Laudato, G., Colavita, A. R., Scalabrino, S., & Oliveto, R. (2022). *A Robust Approach for a Real-time Accurate Screening of ST Segment Anomalies.* In HEALTHINF (pp. 69-80).

C4 Laudato, G., **Rosa, G.**, Capobianco, G., Colavita, A., Dal Forno, A., Divino, F., Lupi, C., Pareschi, R., Ricciardi, S., Romagnoli, L., & others (2022). *Simulating the Doctor's Behaviour: A Preliminary Study on the Identification of Atrial Fibrillation through Combined Analysis of Heart Rate and Beat Morphology.* In HEALTHINF (pp. 446–453).

C3 **Rosa, G.**, Pascarella, L., Scalabrino, S., Tufano, R., Bavota, G., Lanza, M., & Oliveto, R. (2021, May). *Evaluating szz implementations through a developer-informed oracle*. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (pp. 436-447). IEEE.

"Per aspera ad astra" - Through hardships to the stars.

– Latin motto, source unknown

Bibliography

[1] hadolint: Dockerfile linter, validate inline bash, written in haskell. https://github.com/hadolint/hadolint, 2015. [Online; accessed 2-Jun-2022].

[2] A. Author(s). Replication package, TBD. https://figshare.com/s/f988d0be945d30fdff3f.

[3] A. Author(s). Replication package, TBD. https://figshare.com/s/bc4154a81a126878ca54.

[4] B. A. Azad, P. Laperdrix, and N. Nikiforakis. Less is more: quantifying the security benefits of debloating web applications. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1697–1714, 2019.

[5] H. Azuma, S. Matsumoto, Y. Kamei, and S. Kusumoto. An empirical study on self-admitted technical debt in dockerfiles. *Empirical Software Engineering*, 27(2):1–26. Springer, 2022.

[6] P. Becker, M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[7] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal statistical society: series B (Methodological)*, 57(1):289–300. Wiley Online Library, 1995.

[8] A. Brogi, D. Neri, and J. Soldani. Dockerfinder: multi-attribute search of docker images. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 273–278. IEEE, 2017.

[9] Q.-C. Bui, M. Lauköetter, and R. Scandariato. Dockercleaner: Automatic repair of security smells in dockerfiles. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, page To Appear. IEEE, 2023.

[10] N. by IBM. Primerica taps ibm to modernize applications in a hybrid cloud environment. https://www.prnewswire.com/news-releases/primerica-taps-ibm-to-modernize-applications-in-a-hybrid-cloud-environment-300917758.html. [Online; accessed 14-Dec-2023].

[11] 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*. IEEE, 2021.

[12] CIO. Primerica: Ensuring high-quality, modernized code. `https://www.cio.com/article/196255/containers-and-kubernetes-3-transformational-success-stories.html`. [Online; accessed 14-Dec-2023].

[13] 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 *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 323–333. IEEE, 2017.

[14] N. Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin*, 114(3):494. American Psychological Association, 1993.

[15] J. Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46. Sage Publications Sage CA: Thousand Oaks, CA, 1960.

[16] W. Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30. ACM New York, NY, USA, 1992.

[17] O. Dabic, E. Aghajani, and G. Bavota. Sampling projects in github for msr studies. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 560–564. IEEE, 2021.

[18] Docker. Best practices for writing dockerfiles. `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/`. [Online; accessed 2-Jun-2022].

[19] Docker. Docker official images: Review guidelines. `https://github.com/docker-library/official-images#review-guidelines`. [Online; accessed 2-Jun-2022].

[20] Docker. Dockerfile reference. `https://docs.docker.com/engine/reference/builder/`. [Online; accessed 31-Aug-2022].

[21] Docker. Instruction maintainer (deprecated). `https://docs.docker.com/engine/reference/builder/#maintainer-deprecated`. [Online; accessed 15-Dec-2023].

[22] Docker. Docker bench for security. https://github.com/docker/docker-bench-security, 2023. [Online; accessed 16-July-2023].

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

[24] Elastic. The bm25 algorithm and its variables. https://www.elastic.co/blog/practical-bm25-part-2-the-bm25-algorithm-and-its-variables. [Online; accessed 31-Aug-2022].

[25] Elastic. Elasticsearch: The official distributed search and analytics engine. https://www.elastic.co/elasticsearch/. [Online; accessed 31-Aug-2022].

[26] P. D. Ellis. *The essential guide to effect sizes: Statistical power, meta-analysis, and the interpretation of research results*. Cambridge university press, 2010.

[27] K. Eng and A. Hindle. Revisiting dockerfiles in open source software over time. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 449–459. IEEE, 2021.

[28] explosion. spacy: Industrial-strength natural language processing (nlp) in python. https://github.com/explosion/spaCy/commit/d583626, 2017. [Online; accessed 31-Aug-2022].

[29] A. Fink. *The survey handbook*. sage, 2003.

[30] M. Fowler. *Refactoring*. Addison-Wesley Professional, 2018.

[31] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994.

[32] Gartner. 2021 gartner market guide for cloud workload protection platforms. https://businessresources.bitdefender.com/gartner-2021-market-guide-for-cloud-workload-protection-platforms?hs_preview=CPRimYYO-51790146713&hsLang=en-us. [Online; accessed 27-Jun-2022].

[33] A. Gelman and J. Hill. *Data analysis using regression and multilevel/hierarchical models*. Cambridge university press, 2006.

[34] S. Gholami, H. Khazaei, and C.-P. Bezemer. Should you upgrade official docker hub images in production environments? In *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 101–105. IEEE, 2021.

[35] GHTorrent. Contacting users for surveys. https://github.com/ghtorrent/ghtorrent.org/blob/master/faq.md\#contacting-users-for-surveys, 2020. [Online; accessed 14-July-2023].

[36] GitHub. Github copilot: Your ai pair programmer. https://github.com/features/copilot. [Online; accessed 31-Aug-2022].

[37] GoogleContainerTools. container-diff: Diff your docker containers. `https://github.com/GoogleContainerTools/container-diff/commit/0f743be`, 2017. [Online; accessed 31-Aug-2022].

[38] Q. Guo, J. Cao, X. Xie, S. Liu, X. Li, B. Chen, and X. Peng. Exploring the potential of chatgpt in automated code refinement: An empirical study. 2024.

[39] K. Hanayama, S. Matsumoto, and S. Kusumoto. Humpback: Code completion system for dockerfiles based on language models. In *Proc. Workshop on Natural Language Processing Advancements for Software Engineering*, pages 1–4, 2020.

[40] M. U. Haque and M. A. Babar. Well begun is half done: An empirical study of exploitability & impact of base-image vulnerabilities. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1066–1077. IEEE, 2022.

[41] M. U. Haque, L. H. Iwaya, and M. A. Babar. Challenges in docker development: A large-scale study using stack overflow. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2020.

[42] R. He, H. He, Y. Zhang, and M. Zhou. Automating dependency updates in practice: An exploratory study on github dependabot. *IEEE Transactions on Software Engineering*. IEEE, 2023.

[43] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps. A dataset of dockerfiles. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 528–532, 2020.

[44] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps. Learning from, understanding, and supporting devops artifacts for docker. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 38–49. IEEE, 2020.

[45] 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)*, pages 1148–1160. IEEE, 2021.

[46] E. Horton and C. Parnin. Dockerizeme: Automatic inference of environment dependencies for python code snippets. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 328–338. IEEE, 2019.

[47] Y. Huang, D. Ford, and T. Zimmermann. Leaving my fingerprints: Motivations and challenges of contributing to oss for social good. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1020–1032. IEEE, 2021.

[48] Z. Huang, S. Wu, S. Jiang, and H. Jin. Fastbuild: Accelerating docker image building for efficient development and deployment of container. In *2019 35th*

Symposium on Mass Storage Systems and Technologies (MSST), pages 28–37. IEEE, 2019.

[49] M. Hubert and E. Vandervieren. An adjusted boxplot for skewed distributions. *Computational statistics & data analysis*, 52(12):5186–5201. Elsevier, 2008.

[50] J. Humble and D. Farley. Continuous delivery: reliable software releases through build. *Test, and deployment automation. Pearson Education*, 1. 2010.

[51] M. H. Ibrahim, M. Sayagh, and A. E. Hassan. Too many images on dockerhub! how different are images for the same system? *Empirical Software Engineering*, 25(5):4250–4281. Springer, 2020.

[52] Q. Jiang. Improving performance of docker instance via image reconstruction. In *International Conference on Big Data Intelligence and Computing*, pages 511–522. Springer, 2022.

[53] T. W. S. Journal. Adobe turns to devops platform to manage cloud delivery. `https://www.wsj.com/articles/BL-CIOB-7355`. [Online; accessed 14-Dec-2023].

[54] S. Kitajima and A. Sekiguchi. Latest image recommendation method for automatic base image update in dockerfile. In *International Conference on Service-Oriented Computing*, pages 547–562. Springer, 2020.

[55] B. Kitchenham and P. Brereton. A systematic review of systematic review process research in software engineering. *Information and software technology*, 55(12):2049–2075. Elsevier, 2013.

[56] B. A. Kitchenham and S. L. Pfleeger. Personal opinion surveys. *Guide to advanced empirical software engineering*, pages 63–92. Springer, 2008.

[57] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical debt: From metaphor to theory and practice. *Ieee software*, 29(6):18–21. IEEE, 2012.

[58] E. Ksontini, M. Kessentini, T. d. N. Ferreira, and F. Hassan. Refactorings and technical debt in docker projects: An empirical study. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 781–791. IEEE, 2021.

[59] T. Kudo and J. Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*. 2018.

[60] J. R. Landis and G. G. Koch. An application of hierarchical kappa-type statistics in the assessment of majority agreement among multiple observers. *Biometrics*, pages 363–374. JSTOR, 1977.

[61] A. LeClair, S. Jiang, and C. McMillan. A neural model for generating natural language summaries of program subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 795–806. IEEE, 2019.

[62] L. Li, L. Yang, H. Jiang, J. Yan, T. Luo, Z. Hua, G. Liang, and C. Zuo. Auger: Automatically generating review comments with pre-training models. *arXiv preprint arXiv:2208.08014*. 2022.

[63] T.-O. Li, W. Zong, Y. Wang, H. Tian, Y. Wang, S.-C. Cheung, and J. Kramer. Nuances are the key: Unlocking chatgpt to find failure-inducing tests with differential prompting. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 14–26. IEEE Computer Society, 2023.

[64] J. T. Liang, C. Yang, and B. A. Myers. Understanding the usability of ai programming assistants. In *2024 IEEE/ACM 46rd International Conference on Software Engineering (ICSE)*, page To appear. IEEE, 2023.

[65] J. T. Liang, T. Zimmermann, and D. Ford. Understanding skills for oss communities on github. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 170–182, 2022.

[66] C. Lin, S. Nadi, and H. Khazaei. A large-scale data set and an empirical study of docker images hosted on docker hub. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 371–381. IEEE, 2020.

[67] P. Liu, S. Ji, L. Fu, K. Lu, X. Zhang, W.-H. Lee, T. Lu, W. Chen, and R. Beyah. Understanding the security risks of docker hub. In *European Symposium on Research in Computer Security*, pages 257–276. Springer, 2020.

[68] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang. Neural-machine-translation-based commit message generation: how far are we? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 373–384, 2018.

[69] F. Lomio, E. Iannone, A. De Lucia, F. Palomba, and V. Lenarduzzi. Just-in-time software vulnerability detection: Are we there yet? *Journal of Systems and Software*, page 111283. Elsevier, 2022.

[70] Z. Lu, J. Xu, Y. Wu, T. Wang, and T. Huang. An empirical case study on the temporary file smell in dockerfiles. *IEEE Access*, 7:63650–63659. IEEE, 2019.

[71] L. E. Lwakatare, P. Kuvaja, and M. Oivo. Dimensions of devops. In *International conference on agile software development*, pages 212–217. Springer, 2015.

[72] Y. Ma, C. Bogart, S. Amreen, R. Zaretzki, and A. Mockus. World of code: an infrastructure for mining the universe of open source vcs data. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 143–154. IEEE, 2019.

[73] A. Martin, S. Raponi, T. Combe, and R. Di Pietro. Docker ecosystem–vulnerability analysis. *Computer Communications*, 122:30–43. Elsevier, 2018.

[74] R. C. Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[75] A. Mastropaolo, E. Aghajani, L. Pascarella, and G. Bavota. An empirical study on code comment completion. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 159–170. IEEE, 2021.

[76] A. Mastropaolo, E. Aghajani, L. Pascarella, and G. Bavota. Automated variable renaming: are we there yet? *Empirical Software Engineering*, 28(2):45. Springer, 2023.

[77] A. Mastropaolo, N. Cooper, D. N. Palacio, S. Scalabrino, D. Poshyvanyk, R. Oliveto, and G. Bavota. Using transfer learning for code-related tasks. *IEEE Transactions on Software Engineering*. IEEE, 2022.

[78] A. Mastropaolo, L. Pascarella, and G. Bavota. Using deep learning to generate complete log statements. In *Proceedings of the 44th International Conference on Software Engineering*, page 2279–2290. Association for Computing Machinery, 2022.

[79] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 336–347. IEEE, 2021.

[80] V. Nardone, B. Muse, M. Abidi, F. Khomh, and M. Di Penta. Video game bad smells: What they are and how developers perceive them. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–35. ACM New York, NY, USA, 2023.

[81] nodejs.dev. An introduction to the npm package manager. `https://nodejs.dev/en/learn/an-introduction-to-the-npm-package-manager/`, 2022. [Online; accessed 28-July-2023].

[82] D. Nüst and M. Hinz. containerit: Generating dockerfiles for reproducible research with r. *Journal of Open Source Software*, 4(40):1603. 2019.

[83] R. Opdebeeck, J. Lesy, A. Zerouali, and C. De Roover. The docker hub image inheritance network: Construction and empirical insights. In *23rd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2023)*. IEEE, 2023.

[84] OWASP. Owasp docker security cheat sheet. `https://cheatsheetseries.owasp.org/cheatsheets/DockerSecurityCheatSheet.html`, 2023. [Online; accessed 14-July-2023].

[85] P3GLEG. Whaler: Program to reverse docker images into dockerfiles. `https://github.com/P3GLEG/Whaler/commit/d347050`, 2018. [Online; accessed 2-Jun-2022].

[86] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[87] phpdocker io. phpdocker.io: Phpdocker.io website and environment (php and docker based) generator. `https://github.com/phpdocker-io/phpdocker.io/commit/4ea6736`, 2016. [Online; accessed 31-Aug-2022].

[88] D. Pina, C. Seaman, and A. Goldman. Technical debt prioritization: a developer's perspective. In *Proceedings of the International Conference on Technical Debt*, pages 46–55, 2022.

[89] PyPI. The all_packages pip package. `https://pypi.org/project/all-packages/`, 2022. [Online; accessed 28-July-2023].

[90] Qualtrics. Ethical issues to consider when conducting survey research. `https://www.qualtrics.com/blog/ethical-issues-for-online-surveys/`, 2020. [Online; accessed 14-July-2023].

[91] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu, et al. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(140):1–67. 2020.

[92] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel. Cimplifier: automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 476–486, 2017.

[93] V. Rastogi, C. Niddodi, S. Mohan, and S. Jha. New directions for container debloating. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, pages 51–56, 2017.

[94] D. A. Regier, W. E. Narrow, D. E. Clarke, H. C. Kraemer, S. J. Kuramoto, E. A. Kuhl, and D. J. Kupfer. Dsm-5 field trials in the united states and canada, part ii: Test-retest reliability of selected categorical diagnoses. *American Journal of Psychiatry*, 170(1):59–70. 2013. PMID: 23111466.

[95] N. Reimers and I. Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019.

[96] D. Reis, B. Piedade, F. F. Correia, J. P. Dias, and A. Aguiar. Developing docker and docker-compose specifications: A developers' survey. *IEEE Access*, 10:2318–2329. IEEE, 2021.

[97] G. Rosa, A. Mastropaolo, S. Scalabrino, G. Bavota, and R. Oliveto. Automatically generating dockerfiles via deep learning: Challenges and promises. In *2023 IEEE/ACM International Conference on Software and System Processes (ICSSP)*, pages 1–12. IEEE Computer Society, may 2023.

[98] G. Rosa, A. Mastropaolo, S. Scalabrino, G. Bavota, and R. Oliveto. *Replication package*, 2023.

[99] G. Rosa, S. Scalabrino, G. Bavota, and R. Oliveto. *Replication package*, 2023. https://doi.org/10.6084/m9.figshare.20131727.

[100] G. Rosa, S. Scalabrino, G. Bavota, and R. Oliveto. What quality aspects influence the adoption of docker images? *ACM Transactions on Software Engineering and Methodology*. ACM New York, NY, 2023.

[101] G. Rosa, S. Scalabrino, and R. Oliveto. Fixing dockerfile smells: An empirical study. *arXiv preprint arXiv:2208.09097*. 2022.

[102] G. Rosa, F. Zappone, S. Scalabrino, and R. Oliveto. Replication package, TBD. https://figshare.com/s/9f96e94865576421d491.

[103] RubyGems. Ruby gem stats. https://rubygems.org/stats, 2023. [Online; accessed 28-July-2023].

[104] J. Ruscio. A probability-based measure of effect size: robustness to base rates and other factors. *Psychological methods*, 13(1):19. American Psychological Association, 2008.

[105] G. Schermann, S. Zumberi, and J. Cito. Structured information on state and evolution of dockerfiles on github. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 26–29, 2018.

[106] C. Security. Cis docker benchmark. https://www.cisecurity.org/benchmark/docker, 2023. [Online; accessed 14-July-2023].

[107] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar. Trimmer: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 329–339, 2018.

[108] Y. Shin, A. Meneely, L. Williams, and J. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans. Software Eng.*, 37:772–787. 11 2011.

[109] R. Shu, X. Gu, and W. Enck. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 269–280, 2017.

[110] J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. Measuring and modeling programming experience. *Empirical Software Engineering*, 19:1299–1334. Springer, 2014.

[111] D. Skourtis, L. Rupprecht, V. Tarasov, and N. Megiddo. Carving perfect layers out of docker images. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[112] E. Smith, R. Loftin, E. Murphy-Hill, C. Bird, and T. Zimmermann. Improving developer participation rates in surveys. In *2013 6th International workshop on*

cooperative and human aspects of software engineering (CHASE), pages 89–92. IEEE, 2013.

[113] D. Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[114] B. Tak, H. Kim, S. Suneja, C. Isci, and P. Kudva. Security analysis of container images using cloud analytics framework. In *International Conference on Web Services*, pages 116–133. Springer, 2018.

[115] TechBeacon. 10 companies killing it at devops. `https://techbeacon.com/app-dev-testing/10-companies-killing-it-devops`. [Online; accessed 14-Dec-2023].

[116] TechGenix. Containers success stories from 3 leading organizations. `https://techgenix.com/containers-success-stories/`. [Online; accessed 14-Dec-2023].

[117] TheServerSide. How target improved software delivery by adopting devops processes. `https://www.theserverside.com/news/450402724/How-Target-improved-software-delivery-by-adopting-DevOps-processes`. [Online; accessed 14-Dec-2023].

[118] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4):e1838. Wiley Online Library, 2017.

[119] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 403–414. IEEE, 2015.

[120] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, 43(11):1063–1088. IEEE, 2017.

[121] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29. ACM New York, NY, USA, 2019.

[122] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota. Using pre-trained models to boost code review automation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 2291–2302, 2022.

[123] I. Turner-Trauring. Broken by default: why you should avoid most dockerfile examples. `https://pythonspeed.com/articles/dockerizing-python-is-hard/`. [Online; accessed 31-Aug-2022].

[124] D. Van Der Linden, P. Anthonysamy, B. Nuseibeh, T. T. Tun, M. Petre, M. Levine, J. Towse, and A. Rashid. Schrödinger's security: opening the box on app developers' security rationale. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 149–160, 2020.

[125] C. Vassallo, S. Proksch, A. Jancso, H. C. Gall, and M. Di Penta. Configuration smells in continuous delivery pipelines: a linter and a six-month study on gitlab. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 327–337, 2020.

[126] C. Watson, N. Cooper, D. N. Palacio, K. Moran, and D. Poshyvanyk. A systematic literature review on the use of deep learning in software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–58. ACM New York, NY, 2022.

[127] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83. [International Biometric Society, Wiley], 1945.

[128] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[129] Y. Wu, Y. Zhang, T. Wang, and H. Wang. Characterizing the occurrence of dockerfile smells in open-source software: An empirical study. *IEEE Access*, 8:34127–34139. IEEE, 2020.

[130] Y. Wu, Y. Zhang, T. Wang, and H. Wang. An empirical study of build failures in the docker context. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 76–80, 2020.

[131] X. Xia, S. Zhao, X. Zhang, Z. Lou, W. Wang, and F. Bi. Understanding the archived projects on github. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 13–24. IEEE, 2023.

[132] J. Xu, Y. Wu, Z. Lu, and T. Wang. Dockerfile tf smell detection based on dynamic and static analysis methods. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, pages 185–190. IEEE, 2019.

[133] Y. Yao, L. Rosasco, and A. Caponnetto. On early stopping in gradient descent learning. *Constructive Approximation*, 26(2):289–315. Springer, 2007.

[134] H. Ye, J. Zhou, W. Chen, J. Zhu, G. Wu, and J. Wei. Dockergen: A knowledge graph based approach for software containerization. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 986–991. IEEE, 2021.

[135] A. Zerouali, V. Cosentino, T. Mens, G. Robles, and J. M. Gonzalez-Barahona. On the impact of outdated and vulnerable javascript packages in docker images.

In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 619–623. IEEE, 2019.

[136] A. Zerouali, T. Mens, and C. De Roover. On the usage of javascript, python and ruby packages in docker hub images. *Science of Computer Programming*, 207:102653. Elsevier, 2021.

[137] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona. On the relation between outdated docker containers, severity vulnerabilities, and bugs. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 491–501. IEEE, 2019.

[138] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric. Coditt5: Pretraining for source code and natural language editing. *arXiv preprint arXiv:2208.05446*. 2022.

[139] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6):1245–1262. SIAM, 1989.

[140] Y. Zhang, B. Vasilescu, H. Wang, and V. Filkov. One size does not fit all: an empirical study of containerized continuous deployment workflows. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 295–306, 2018.

[141] Y. Zhang, H. Wang, and V. Filkov. A clustering-based approach for mining dockerfile evolutionary trajectories. *Science China Information Sciences*, 62(1):1–3. Science China Press, 2019.

[142] Y. Zhang, G. Yin, T. Wang, Y. Yu, and H. Wang. An insight into the impact of dockerfile evolutionary trajectories on quality and latency. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, pages 138–143. IEEE, 2018.

[143] Y. Zhang, Y. Zhang, X. Mao, Y. Wu, B. Lin, and S. Wang. Recommending base image for docker containers based on deep configuration comprehension. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 449–453. IEEE, 2022.

[144] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. K. Paul, K. Chen, and A. R. Butt. Large-scale analysis of docker images and performance implications for container storage systems. *IEEE Transactions on Parallel and Distributed Systems*, 32(4):918–930. IEEE, 2020.

[145] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. S. Warke, M. Mohamed, and A. R. Butt. Large-scale analysis of the docker hub

dataset. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10. IEEE, 2019.

[146] Y. Zhou, W. Zhan, Z. Li, T. Han, T. Chen, and H. Gall. Drive: Dockerfile rule mining and violation detection. *arXiv preprint arXiv:2212.05648*. 2022.

[147] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering*, 47(10):2084–2106. IEEE, 2019.