Noname manuscript No.
(will be inserted by the editor)

Fixing Dockerfile Smells: An Empirical Study

Giovanni Rosa · Federico Zappone · Simone Scalabrino · Rocco Oliveto

Received: date / Accepted: date

Abstract Docker is the *de facto* standard for software containerization. A Dockerfile contains the requirements to build a Docker image containing a target application. There are several best practice rules for writing Dockerfiles, but the developers do not always follow them. Violations of such practices, known as Dockerfile smells, can negatively impact the reliability and performance of Docker images. Previous studies showed that Dockerfile smells are widely diffused, and there is a lack of automatic tools that support developers in fixing them. However, it is still unclear what Dockerfile smells get fixed by developers and to what extent developers would be willing to fix smells in the first place. The aim of our study is twofold. First, we want to understand what Dockerfiles smells receive more attention from developers, *i.e.,* are fixed more frequently in the history of open-source projects. Second, we want to check if developers are willing to accept changes aimed at fixing Dockerfile smells (*e.g.,* generated by an automated tool), to understand if they care about them. We evaluated the survivability of Dockerfile smells from a total of 53,456 unique Dockerfiles, where we manually validated a large sample of smell-removing commits to understand (i) if developers performed the change with the intention of removing bad practices, and (ii) if they were aware of the removed smell.

G. Rosa
STAKE Lab, University of Molise, Italy
E-mail: giovanni.rosa@unimol.it

F. Zappone
STAKE Lab, University of Molise, Italy
E-mail: f.zappone1@studenti.unimol.it

S. Scalabrino
STAKE Lab, University of Molise, Italy
E-mail: simone.scalabrino@unimol.it

R. Oliveto
STAKE Lab, University of Molise, Italy
E-mail: rocco.oliveto@unimol.it

In the second part, we used a rule-based tool to automatically fix Dockerfile smells. Then, we proposed such fixes to developers via pull requests. Finally, we quantitatively and qualitatively evaluated the outcome after a monitoring period of more than 7 months. The results of our study showed that most developers pay more attention to changes aimed at improving the performance of Dockerfiles (image size and build time). Moreover, they are willing to accept the fixes for the most common smells, with some exceptions (*e.g.,* missing version pinning for OS packages).

Keywords dockerfile smells · empirical software engineering · software evolution

1 Introduction

Software systems are developed to be deployed and used. Operating software in a production environment, however, entails several challenges. 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[1]. Among the others, Docker[2] is one of the most popular platforms used in the DevOps workflow: It is the main containerization framework in the open-source community [6], and is widely used by professional developers[3]. Also, Docker is the most loved and most wanted platform in the 2021 StackOverflow survey[3]. Docker allows releasing applications together with their dependencies through containers (*i.e.,* virtual environments) sharing the host operating system kernel. Each Docker image is defined through a Dockerfile, which contains instructions to build the image containing the application. All the public Docker images are hosted on an online repository called DockerHub[4]. Since its introduction in 2013, Docker counts 3.3M of Desktop installations, and 318B image pulls from DockerHub[5].

Defining Dockerfiles, however, is far from trivial: Each application has its own dependencies and requires specific configurations for the execution environment. Previous work [21] introduced the concept of Dockerfile smells, which are violations of best practices, similarly to code smells [5], and a catalog of such problems[6]. The presence of such smells might increase the risk of build failures, generate oversized images, and security issues [6, 10, 22, 23]. Previous work studied the prevalence of Dockerfile smells [6, 9, 14].

Despite the popularity and adoption of Docker, there is still a lack of tools to support developers in improving the quality and reliability of containerized

[1] https://portworx.com/blog/2017-container-adoption-survey/
[2] https://www.docker.com/
[3] https://insights.stackoverflow.com/survey/2021
[4] https://hub.docker.com/
[5] https://www.docker.com/company/
[6] https://github.com/hadolint/hadolint/wiki

applications, *e.g.,* tools for automatic refactoring of code smells on Dockerfiles [13]. Relevant studies in this area investigated the prevalence of Dockerfile smells in open-source projects [6,9,14,21], the diffusion technical debt [4], and the refactoring operations typically performed by developers [13].

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.* [9] 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 paper, we propose a study to fill 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 [2]. The tool performs a rule check on a parsed Abstract Syntax Tree (AST) representation of the input Dockerfile, based on the Docker [1] and shell script [3] 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 the Docker guidelines[7]. 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 provided with our study are the following:

1. We performed a detailed analysis of the survivability of Dockerfile smells and manually validated a sample of smell-fixing commits for Dockerfile smells;
2. We introduced DOCKLEANER, a rule-based tool to fix the most common Dockerfile smells;
3. We ran an evaluation via pull requests of the willingness of developers of accepting changes aimed at fixing Dockerfile smells.

The remaining of the paper is organized as follows: In Section 2 we provide a general overview on Dockerfile smells and related works. Section 3 describes the design of our study, while in Section 5 we present the results of our experiment. In section Section 6 we qualitatively discuss the results. Finally, Section 7 discusses the threats to validity and in Section 8 we summarize some final remarks and future directions.

2 Background and Related Work

Technical debt [12] has a negative impact on the software maintainability. A symptom of technical debt is represented by *code smells* [5]. Code smells are poor implementation choices, that does not follow design and coding best practices, such as design patterns. They can negatively impact the maintainability of the overall software system. Mainly, code smells are defined for object-oriented systems. Some examples are duplicated code or god class (*i.e.,* a class having too much responsibilities). In the following, we first introduce smells that affect Dockerfile, and then we report recent studies on their diffusion and the practices used to improve Dockerfile quality.

Dockerfile smells. Docker reports an official list of best practices for writing Dockerfiles [1]. Such best practices also include indications for writing shell script code included in the `RUN` instructions of Dockerfiles. For example, the usage of the instruction `WORKDIR` instead of the bash command `cd` to change directory. This because each Docker instruction defines a new layer at the time

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

of build. The violation of such practices lead to the introduction of Dockerfile smells. In fact, with Dockerfile smells, we indicate that instructions of a Dockerfile that violate the writing best practices and thus can negatively affect the quality of them [21]. The presence of Dockerfile smells can also have a direct impact on the behavior of the software in a production environment. For example, previous work showed that missing adherence to best practices can lead to security issues [22], negatively impact the image size [10], increase build time and affect the reproducibility of the final image (*i.e.,* build failures) [6, 10, 23]. For example, the version pinning smell, that consists in missing version number for software dependencies, can lead to build failures as with dependencies updates the execution environment can change. There are several tools that support developers in writing Dockerfiles. An example is the *binnacle* tool, proposed by Henkel *et al.* [10] that performs best practices rule checking defined on the basis of a dataset of Dockerfiles written by experts. The reference tool used in literature for the detection of Dockerfile smells is *hadolint* [2]. Such a tool checks a set of best practices violations on a parsed AST version of the target Dockerfile using a rule-based approach. Hadolint detects two main categories of issues: Docker-related and shell-script-related. The former affect Dockerfile-specific instructions (*e.g.,* the usage of absolute path in the `WORKDIR` command[8]). They are identified by a name having the prefix *DL* followed by a number. The shell-script-related violations, instead, specifically regard the shell code in the Dockerfile (*e.g.,* in the `RUN` instructions). Such violations are a subset of the ones detected by the *ShellCheck* tool [3] and they are identified by the prefix *SC* followed by a number. It is worth saying that these rules can be updated and changed during time. For example, as the instruction `MAINTAINER` has been deprecated, the rule DL4000 that previously check for the usage of that instructions that was a best practice, has been updated as the avoidance of that instruction because it is deprecated.

Diffusion of Dockerfile smells. A general overview of the diffusion of Dockerfile smells was proposed by Wu *et al.* [21]. 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 that the shell-script smells. Also in this direction, Cito *et al.* [6] performed an empirical study to characterize the Docker ecosystem in terms of quality issues and evolution of Dockerfiles. They found that the most frequent smell regards the lack of version pinning for dependencies, that can lead to build fails. Lin *et al.* [14] 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 smells prevalence over time, where the version pinning

[8] `https://github.com/hadolint/hadolint/wiki/DL3000`

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. Eng *et al.* [9] conducted an empirical study on the largest dataset of Dockerfiles, spanning from 2013 to 2020 and having over 9.4 million unique instances. They performed an historical analysis on the evolution of Dockerfiles, reproducing the results of previous studies on their 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 at the fact that developers might be interested in fixing them. Still, it is unclear the reason behind their disappearance, *e.g.,* if developers actually fix them or if they get removed incidentally.

3 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:

- **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).

3.1 Study Context

The context of our study is represented by a subset of the dataset introduced by Eng *et al.* [9]. 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 [6,10,13]. 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 [15].

3.2 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

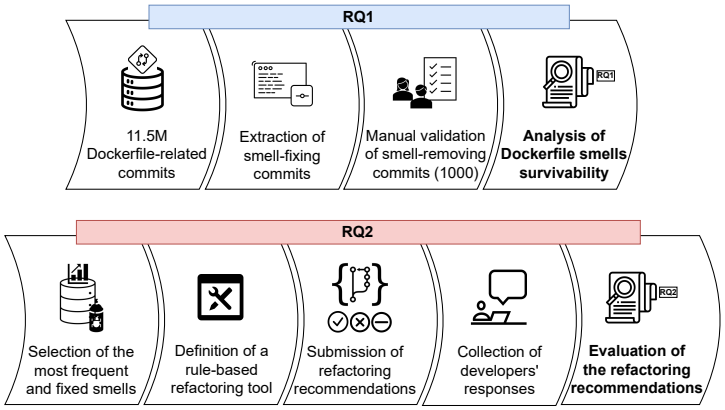


Fig. 1: Overall workflow of the experimentation procedure.

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*[9]. For each valid Dockerfile, we mined the change 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*[10] for each Dockerfile to extract the Dockerfile smells, if present.

4 Experimental Procedure

In this section, we describe the experimentation procedure that we will use to answer our RQs. Fig. 1 describes the overall workflow of the study.

4.1 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

[9] `https://github.com/asottile/dockerfile`
[10] hadolint release v2.12.0

```
RUN apt-get install -y \
    curl=7.* \
    wget \
    && rm -rf /var/lib/apt/lists/*
```

```
RUN apt-get install -y \
    curl=7.* \

    && rm -rf /var/lib/apt/lists/*
```

Fig. 2: Example of a candidate smell-fixing commit that does not actually fix the smell.

respective commit IDs in which they were introduced (*i.e.,* $c(d_1), \ldots, c(d_n)$). We also 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 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. 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.

Table 1: The most frequent Dockerfile smells identified in literature [9], 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[11]

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 [7], obtaining a value of $k = 0.79$ considered "*very good*" according to the interpretation recommendations [16]. 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 [19], 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 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 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[12]. Next, we use DOCKLEANER to fix smells in existing Dockerfiles from open-source projects and submit the changes to the 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.

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

4.2.1 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 [9]. 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 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.* [11]. Specifically, to determine the correct image tag, we use the 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[13]. 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. 3.

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

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

Fig. 3: Example of rule DL3006.

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 [9]. 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` 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. 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

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

Fig. 4: Example of rule DL3008.

H i!
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.

Fig. 5: Example of the pull request message. The placeholders (wrapped in curly braces) will be replaced with the corresponding values.

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.

4.2.2 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 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. To do this, we defined and used a structured template for all the pull requests, as reported in Fig. 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.* [20]. 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

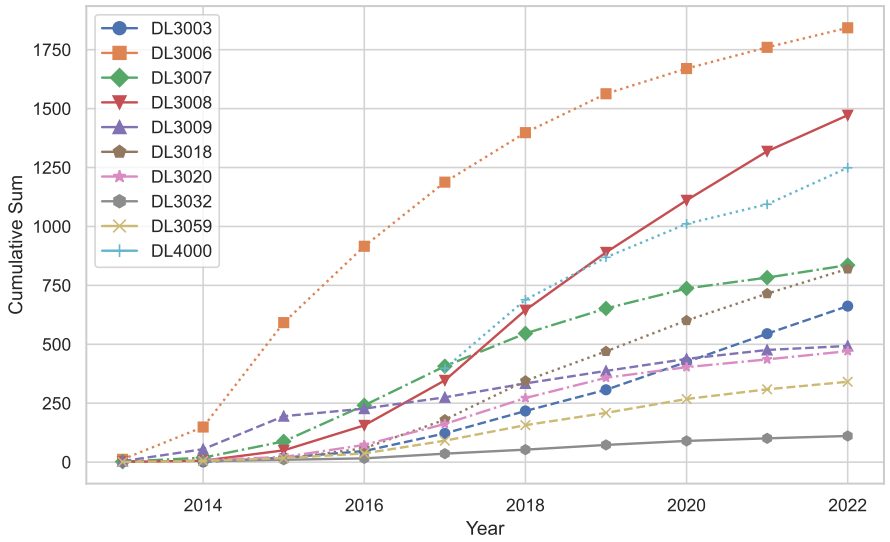


Fig. 6: Occurrence over time for the top 10 Dockerfile smells.

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 [18] 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.

4.2.3 Data Availability

The code and data used in our study, along with the implementation of DockLeaner, can be found in the replication package [17].

5 Analysis of the Results

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

5.1 RQ$_1$: *How do developers fix Dockerfile smells?*

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

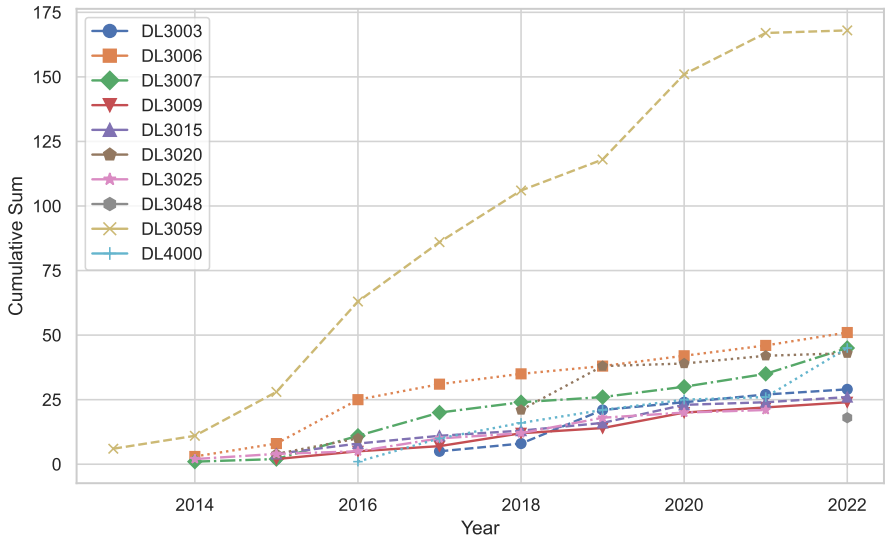


Fig. 7: Fixing trend over time for the 10 most fixed Dockerfile smells.

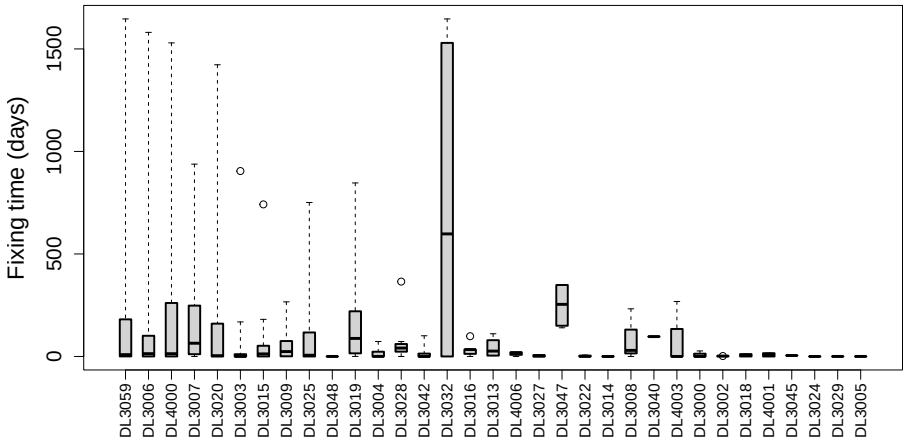


Fig. 8: Overall fixing time delta (days) among all Dockerfile smells.

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[14] after the deprecation of the `MAINTAINER` instruction, we excluded its occurrences before that date from the plot.

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

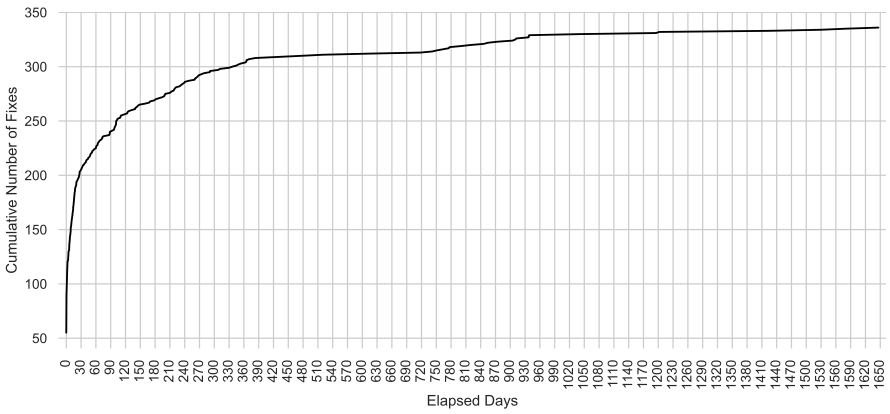


Fig. 9: Cumulative fixes over time interval (days) among all Dockerfile smells.

Table 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

In our manual validation, we found that 33.6% of the commits in which smells disappear actually fix smells. We report in Table 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. 8 the adjusted boxplots describing the days that passed after each smell got fixed. We report in Fig. 7 the fixing trend 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. 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 the other hand, if a smell survives the first day, it is less likely that it gets fixed later. In fact, according to Table 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 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`[15]. 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.

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[16] and the number of layers[17]. In some cases, we found that developers use linters to detect bad practices. Among those, only one commit explicitly mentioned *hadolint*[18], while in other cases they mentioned the tool *DevOps-Bash-tools*[19].

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

[15] `https://github.com/webbertakken/K8sSymfonyReact/commit/811582f`
[16] `https://github.com/KDE/kaffeine/commit/d03145b`
[17] `https://github.com/Eadom/ctf_xinetd/commit/21f2785`
[18] `https://github.com/flyway/flyway-docker/commit/3eeabe5`
[19] `https://github.com/HariSekhon/Dockerfiles/commit/eeab92a`

Table 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

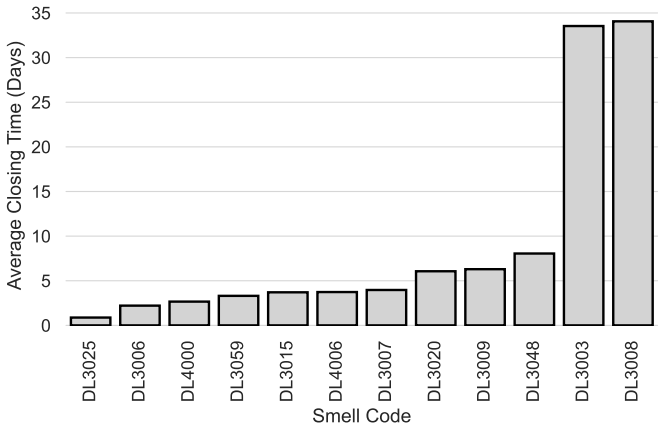
because they are more interested in the optimization of other non-functional aspects such as build time and size of the Docker image.

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 34% 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).

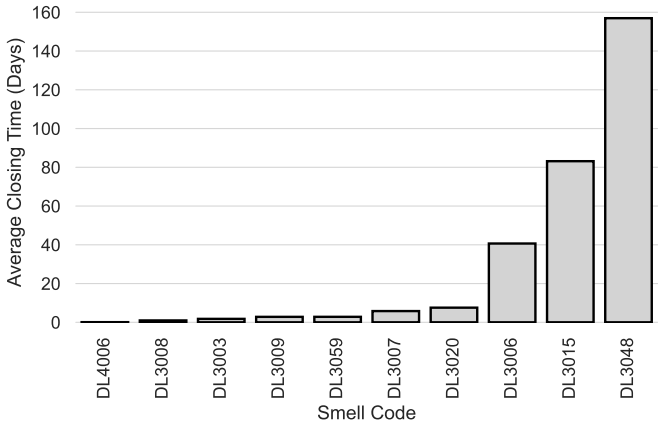
5.2 RQ$_2$: *Which Dockerfile smells are developers willing to address?*

In Table 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 `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



(a)



(b)

Fig. 10: Average resolution time (days) for merged pull requests (a) and rejected pull requests (b).

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. 11 we report the adjusted boxplot for the time required for pull requests to get the first response and to be resolved. Additionally, Fig. 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

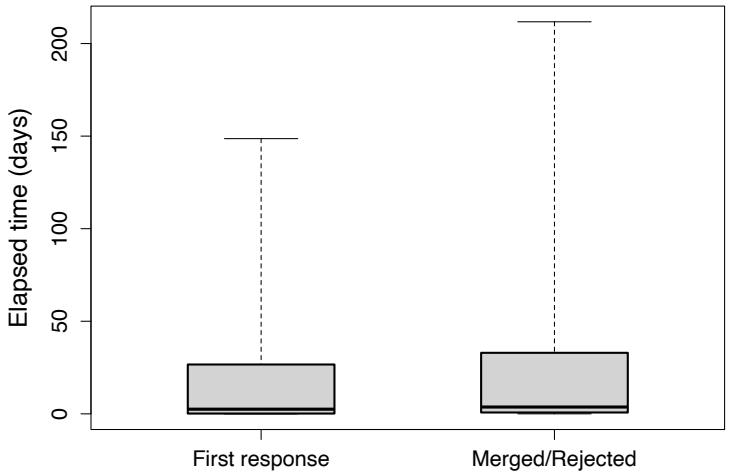


Fig. 11: Adjusted boxplot of the number of days required for a pull request to obtain a response (left) and to be merged/rejected (right).

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 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 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).

Table 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

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.

6 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, 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[20], 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`[21], 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).

[20] `https://github.com/Shopify/semian/pull/484`
[21] `https://github.com/envoyproxy/ratelimit/pull/411`

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[22], or it might make the build more fragile[23]. 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[24]. 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[25]. 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.

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`[26] (DL3020) and `320ba87`[27] (DL4006). An interesting discussion arose for the rejected fix of DL3003[28]. 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. 12, where the fix has been rejected because the change of the working directory is temporary.

[22] https://github.com/James-Yu/LaTeX-Workshop/pull/3837
[23] https://github.com/Yelp/aactivator/pull/47
[24] https://github.com/FDio/govpp/pull/123
[25] https://github.com/Lookyloo/lookyloo/pull/663
[26] https://github.com/ROCmSoftwarePlatform/Tensile/pull/1707
[27] https://github.com/bupy7/xml-constructor/pull/6
[28] https://github.com/NUbots/NUbots/pull/1063

```
@@ -53,7 +53,8 @@ COPY etc/ld.so.conf.d/usrlocal.conf /etc/ld.so.conf.d/usrlocal.conf
53  53  RUN ldconfig
54  54
55  55  # Make a symlink from /usr/local/lib to /usr/local/lib64 so library install location is irrelevant
56      - RUN cd /usr/local && ln -sf lib lib64
    56  + WORKDIR /usr/local
    57  + RUN  ln -sf lib lib64
57  58
58  59  # Generate toolchain files for the generic platform
59  60  COPY usr/local/toolchain/generate_toolchains.py /usr/local/generate_toolchains.py
```

Fig. 12: Example of a wrong fix for DL3003. In that case, the change of working directory is temporary, and the fix has been rejected.

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.

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[29]. 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)[30], 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

[29] `https://github.com/riga/law/pull/152`
[30] `https://github.com/lablup/backend.ai/pull/1216`

defined by *opencontainers* [31], which is also suggested for DL4000 in the official Docker documentation [32]. The second is more general and does not enforce a pre-defined format. It is reported in the *hadolint* documentation [33], which also is reported in the official Docker documentation as examples of `LABEL` instructions [34]. The fixes that we analyzed in $\mathbf{RQ_1}$ that follow the first format are limited only to one repository[35]. In other cases, developers adopted the second format[36]. 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[37].

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[38]. 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`. In the example reported in Fig. 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).

[31] https://specs.opencontainers.org/image-spec/annotations/
[32] https://docs.docker.com/engine/reference/builder/#maintainer-deprecated
[33] https://github.com/hadolint/hadolint/wiki/DL3048
[34] https://docs.docker.com/engine/reference/builder/#label
[35] https://github.com/HariSekhon/Dockerfiles/commit/f329b94
[36] https://github.com/scossu/lakesuperior/commit/a552ff7
[37] https://github.com/hpc/charliecloud/pull/1628
[38] https://github.com/hpc/charliecloud/commit/aae89d7

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.

7 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*[39]. We choose *hadolint* which is commonly used in the literature [6, 9, 14, 21] and also in enterprise tools for code quality[40]. However, *hadolint* could lead to false positives or can miss some smells[41]. 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.* [9] by considering only those having stargazers count greater or equal to 10. This is commonly used in the literature to avoid toy projects [8]. 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.* [9]. 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 DockCleaner, 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

[39] `https://github.com/goodwithtech/dockle`
[40] `https://github.com/codacy/codacy-hadolint`
[41] `https://github.com/hadolint/hadolint/issues/693`

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 in a different way.

8 Conclusion

In the last few years, containerization technologies have had a significant impact on the deployment workflow. Best practice violations, namely Dockerfile smells, are widely spread in Dockerfiles [6, 9, 14, 21]. In our empirical study, we evaluated 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 also provide several lessons learned that could guide future research in this field and help practitioners in handling Dockerfile smells.

Acknowledgements The work by Rocco Oliveto, Giovanni Rosa, and Simone Scalabrino was supported by the European Union - NextGenerationEU through the Italian Ministry of University and Research, Projects PRIN 2022 "QualAI: Continuous Quality Improvement of AI-based Systems", grant n. 2022B3BP5S, CUP: H53D23003510006.

The authors would like to thank Alessandro Giagnorio (University of Molise, Italy) for implementing a preliminary version of Dockleaner.

Conflict of interest

The authors declare that they have no conflict of interest.

References

1. Best practices for writing Dockerfiles. [Online; accessed 2-Jun-2022]
2. hadolint: Dockerfile linter, validate inline bash, written in Haskell. [Online; accessed 28-May-2022]
3. ShellCheck, a static analysis tool for shell scripts. [Online; accessed 2-Jun-2022]
4. Azuma, H., Matsumoto, S., Kamei, Y., Kusumoto, S.: An empirical study on self-admitted technical debt in dockerfiles. Empirical Software Engineering **27**(2), 1–26 (2022)
5. Becker, P., Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional (1999)
6. Cito, J., Schermann, G., Wittern, J.E., Leitner, P., Zumberi, S., Gall, H.C.: An empirical analysis of the docker container ecosystem on github. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 323–333. IEEE (2017)
7. Cohen, J.: A coefficient of agreement for nominal scales. Educational and psychological measurement **20**(1), 37–46 (1960)
8. Dabic, O., Aghajani, E., Bavota, G.: Sampling projects in github for msr studies. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pp. 560–564. IEEE (2021)
9. Eng, K., Hindle, A.: Revisiting dockerfiles in open source software over time. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pp. 449–459. IEEE (2021)
10. Henkel, J., Bird, C., Lahiri, S.K., Reps, T.: Learning from, understanding, and supporting devops artifacts for docker. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), pp. 38–49. IEEE (2020)
11. Kitajima, S., Sekiguchi, A.: Latest image recommendation method for automatic base image update in dockerfile. In: International Conference on Service-Oriented Computing, pp. 547–562. Springer (2020)
12. Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: From metaphor to theory and practice. Ieee software **29**(6), 18–21 (2012)
13. Ksontini, E., Kessentini, M., Ferreira, T.d.N., Hassan, F.: Refactorings and technical debt in docker projects: An empirical study. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 781–791. IEEE (2021)
14. Lin, C., Nadi, S., Khazaei, H.: 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), pp. 371–381. IEEE (2020)
15. Ma, Y., Bogart, C., Amreen, S., Zaretzki, R., Mockus, A.: 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), pp. 143–154. IEEE (2019)
16. Regier, D.A., Narrow, W.E., Clarke, D.E., Kraemer, H.C., Kuramoto, S.J., Kuhl, E.A., Kupfer, D.J.: 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). DOI 10.1176/appi.ajp.2012.12070999. URL `https://doi.org/10.1176/appi.ajp.2012.12070999`. PMID: 23111466
17. Rosa, G., Zappone, F., Scalabrino, S., Oliveto, R.: Replication package (2024). `https://doi.org/10.6084/m9.figshare.23522679`
18. Spencer, D.: Card sorting: Designing usable categories. Rosenfeld Media (2009)
19. Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D.: When and why your code starts to smell bad (and whether the smells go away). IEEE Transactions on Software Engineering **43**(11), 1063–1088 (2017)

20. Vassallo, C., Proksch, S., Jancso, A., Gall, H.C., Di Penta, M.: 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, pp. 327–337 (2020)
21. Wu, Y., Zhang, Y., Wang, T., Wang, H.: Characterizing the occurrence of dockerfile smells in open-source software: An empirical study. IEEE Access **8**, 34127–34139 (2020)
22. Zerouali, A., Mens, T., Robles, G., Gonzalez-Barahona, J.M.: On the relation between outdated docker containers, severity vulnerabilities, and bugs. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 491–501. IEEE (2019)
23. Zhang, Y., Vasilescu, B., Wang, H., Filkov, V.: 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, pp. 295–306 (2018)