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

Giovanni Rosa
STAKE Lab, University of Molise
Pesche, Italy
giovanni.rosa@unimol.it

Simone Scalabrino
STAKE Lab, University of Molise
Pesche, Italy
simone.scalabrino@unimol.it

Gregorio Robles
Universidad Rey Juan Carlos
Madrid, Spain
grex@gsyc.urjc.es

Rocco Oliveto
STAKE Lab, University of Molise
Pesche, Italy
rocco.oliveto@unimol.it

ABSTRACT

Dockerfiles can be affected by bad design choices, known as *Dockerfile smells*. *Hadolint* is currently the reference tool able to detect them, and it is widely used both by researchers and practitioners. The literature shows that these smells are commonly diffused in Dockerfiles, but it is still not clear how developers perceive them as bad practices. This paper aims to investigate the relevance of the Dockerfile smells captured by *hadolint* from the perspective of expert Dockerfile developers. We first perform a mining study in which we extract the change history of Dockerfiles maintained by experts to understand what smells have been more frequently introduced in their history. Next, we ran a survey in which we asked expert Dockerfile developers to evaluate Dockerfiles affected by different smells. We obtained 94 responses for 17 smells, representative of 24 Dockerfile smells. We found that experts prioritize a small part of the evaluated smells over others. Besides, they report additional bad practices not mapped as smells in any existing catalog. Thus, we propose a ranked catalog containing 26 additional Dockerfile smells, which can be used as a guide for novices to understand which aspects to focus on to write good-quality Dockerfiles.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**.

KEYWORDS

Empirical Software Engineering, Docker, Code Smells

ACM Reference Format:
Giovanni Rosa, Simone Scalabrino, Gregorio Robles, and Rocco Oliveto. 2018. Not all Dockerfile Smells are the Same: An Empirical Evaluation of Hadolint Writing Practices by Experts. In *Proceedings of 21st International Conference on Mining Software Repositories (MSR 2024).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

MSR 2024, April 2024, Lisbon, Portugal
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Docker[1] is the most popular containerization platform.[2] Its main purpose is to encapsulate software applications along with dependencies and configurations in a lightweight isolated environment. This ensures portability, fast deployment, and a lower degree of variability between testing and production environments. Docker containers are based on images which are built from Dockerfiles. Dockerfiles contain instructions written in a domain-specific language[3] that specify the actions to perform to set up the environment (*e.g.,* install dependencies).

Similarly to source code, Dockerfiles can be affected by bad design choices. For the source code, these are often referred to as *code smells* [16]. Code smells can lead to technical debt, impacting negatively on the quality of a software system in terms of maintainability [34]. Code smells for Dockerfiles, also known as *Dockerfile smells*, can negatively impact the quality of Dockerfiles [13].

Several tools have been proposed in the literature to detect Dockerfile smells [8, 9, 20]. *Hadolint* [1] is currently the reference tool used in practice and by researchers to assess the best writing practices [13, 14, 26]. Such a tool checks for more than 66 rules capturing the best writing practices based on the official Docker guidelines [6] and the recommendations of the developer community.

The research literature has investigated how to identify Dockerfile smells and how often they appear [13, 14, 26]. 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. Eng *et al.* [14] 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 [20]. This brings up a natural question, still unanswered: *How do expert developers perceive the Dockerfile smells?* 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 currently available catalog of smells.

[1] https://www.docker.com/
[2] https://survey.stackoverflow.co/2023/#section-most-popular-technologies-other-tools
[3] https://docs.docker.com/engine/reference/builder/

In this paper, we aim at answering such a question. Specifically, we focus on the smells captured by hadolint, the reference tool used both in research [13, 14, 26] and in practice. 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 some 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 our paper is organized as follows. In Section 2 we give some concepts on Docker, and we report related works from the literature. Section 3 presents the results of a mining study to understand if Dockerfiles written by experts adhere to best writing practices. In Section 4 we report a survey to understand how experienced developers perceive Dockerfile smells. In Section 5 we discuss the results, followed by the proposal of an enhanced catalog of Dockerfile smells in Section 6. Section 7 presents the threats to validity. Finally, in Section 8 we summarize the conclusion along with future directions.

2 BACKGROUND AND RELATED WORK

In this section, we describe some basic concepts of Docker and we report existing studies about quality and writing practices for Dockerfiles.

2.1 Docker Basics

Dockerfiles are used to specify the dependencies and the environment to containerize a specific software application. A Dockerfile can extend a *base image*, *i.e.,* an existing Docker image. Dockerfiles are composed of instructions written in a Domain Specific Language (DSL). Each instruction is composed of a Docker-specific keyword (*e.g.,* `RUN`) and one or more arguments. The build process generates Docker images from Dockerfiles. Docker images are composed of stacked *layers*, each of them corresponding to the result of the execution of an instruction. Docker uses a layer caching system which allows reusing previously built layers to optimize successive builds of the same Dockerfile.

Docker images are "pulled" and distributed through *registries*, which can be publicly accessible (*e.g.,* DockerHub[4]) or private with restricted access. Usually, each Docker image is usually assigned with a *tag*, indicating the specific version or flavor in the registry. DockerHub offers a group of selected images labeled as *official images*[5] that are directly maintained by Docker experts [6].

2.2 Dockerfile Smells

Dockerfile smells are violations of best practices [36]. *Hadolint* is the most popular tool for detecting Dockerfile smells, and also widely used in the literature[13, 14, 26]. The rules enforced by the tool have been defined by the community based on (i) the official Dockerfile guidelines provided by Docker, and (ii) the suggestions of Dockerfile developers (*i.e.,* by submitting pull requests[6]).

More in detail, the tool detects two categories of violations: (i) errors in Dockerfile instructions, and (ii) *shell-script* errors. All the *hadolint* 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 documentation [6].

Cito *et al.* [13] were the first to investigate the diffusion of Dockerfile smells. Their results show that smells are diffused in open-source Dockerfiles, where most of the quality issues (28.6%) arise from missing version pinning. Wu *et al.* [36] evaluated more than 6k projects reporting the most occurring Dockerfile smells. Their results show that 84% of GitHub projects contain Dockerfiles affected by Dockerfile smells. Lin *et al.* [26] performed an extended evaluation on the quality of Docker images from DockerHub, investigating also aspects related to the evolution. They report that, even if Dockerfile smells are widely diffused, there is a decreasing trend over time, suggesting that developers are more and more aware of them. Eng *et al.* [14] proposed an updated version of that study, extended to 9.4M Dockerfiles, confirming the previous findings.

Previous studies in the literature proposed catalogs of Dockerfile smells and detection approaches, different from *hadolint*. Henkel *et al.* [20] proposed the *binnacle* tool, which checks for a different set of Dockerfile smells extracted by a rule-mining approach executed directly on the official Dockerfiles written by Docker maintainers. 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 when using *wget* to download external sources. This could lead to security issues. An extended set of these rules is proposed by Zhou *et al.* [38]. They introduced *DRIVE*, an approach for rule mining and smell detection in Dockerfiles supporting a total of 34 semantic and 19 syntactic rule violations. Among those, 9 semantic rules are newly introduced and not supported by the previous tools. Finally, Bui *et al.* [12] proposed DockerCleaner, an approach to detect security-related best practices, based on the gray literature [7, 10]. Some of the considered smells are in overlap with those included in *hadolint.*

[4] https://hub.docker.com/
[5] https://docs.docker.com/docker-hub/official_images/
[6] https://github.com/hadolint/hadolint/pull/114

In our study, we considered only the writing rules from the catalog defined by the *hadolint* community. 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. Other, similar tools, such as Binnacle[7] and DRIVE[8] are not as well-known by the community (*i.e.,* low number of stars), and even if their source code is publicly available on GitHub, they are still far away from being easy to adopt by practitioners. Hadolint aims to adhere to the best writing practices recommended by the official Docker documentation [6], defined directly by the developer community. Its writing practices are more oriented towards the composition of Dockerfiles, which is what we are interested in, while the catalogs of Binnacle and DRIVE contain mostly semantic rules for scripting 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 [19] or DRIVE [38].

Ksontini *et al.* show that the available smells are not the main target of refactoring operations performed on Dockerfiles [23], and Rosa *et al.* show that they do not explain the adoption of Docker images [29]. This means that the existing catalogs might not be comprehensive of the observable smells. To the best of our knowledge, this is the first mining-based study that validates the relevance of Dockerfile smells by asking developers, in particular those who have high expertise in the field, and reporting their feedbacks.

3 STUDY 1: DOCKERFILE SMELLS AFFECTING THE CODE WRITTEN BY EXPERTS

Our 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 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.* [20]. In detail, the repositories contain Dockerfiles created and maintained by experienced developers from Docker, *i.e.,* `docker-library`,[9], that are part of the *Docker official images program.*[10] 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.

Table 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

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 [30].

3.2 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[11] 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.

3.3 Empirical Study Results

We report in Table 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 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.

[7] https://github.com/jjhenkel/binnacle-icse2020
[8] https://github.com/zwlin98/DRIVE
[9] https://github.com/docker-library/
[10] https://github.com/docker-library/official-images#what-are-official-images
[11] https://github.com/asottile/dockerfile

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 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 directory), along with missing version pinning (DL3008 and DL3018) are among the most occurring ones.

4 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.

4.1 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.

4.1.1 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 because the Docker community has few expert developers compared to other developer communities [17].

We opted for a sampling strategy that consists of the selection of contributors from GitHub repositories, which is a common practice in the literature [18, 21, 24, 25, 37, 39]. In detail, we perform a *convenience sampling* [15] 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.* [26]. 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 [2, 3]. 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 [24, 25, 28, 32, 35].

4.1.2 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 pinning 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.

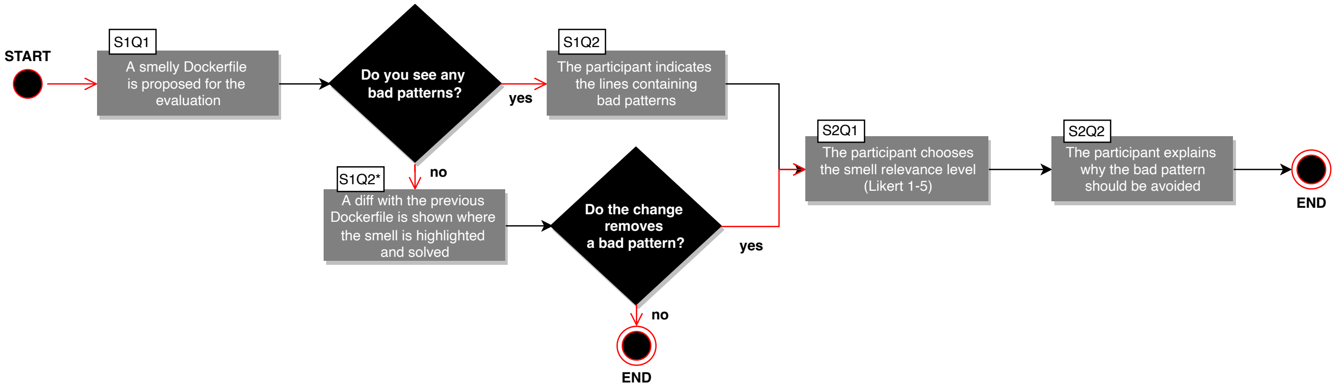
START
S1Q1
A smelly Dockerfile is proposed for the evaluation
Do you see any bad patterns?
yes
S1Q2
The participant indicates the lines containing bad patterns
no
S1Q2*
A diff with the previous Dockerfile is shown where the smell is highlighted and solved
Do the change removes a bad pattern?
yes
no
END
S2Q1
The participant chooses the smell relevance level (Likert 1-5)
S2Q2
The participant explains why the bad pattern should be avoided
END

Figure 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).

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

4.2 Experimental Procedure

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

4.2.1 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:

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 [27].

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. 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 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 2 all the questions contained in the survey for each step.

[12] `npm` counted more than 2.1M packages in 2022 [5], while `pip` had ~350k in the same year [4] and `gem` has 178k packages in 2023 [11].

Table 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 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. The Dockerfiles that we propose in each task are available in our replication package [30].

4.3 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 [22, 24]. 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).

4.4 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$_2$.

4.4.1 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.* [31]. When asked about their overall programming experience, 23 of the responses are from those with 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. 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 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

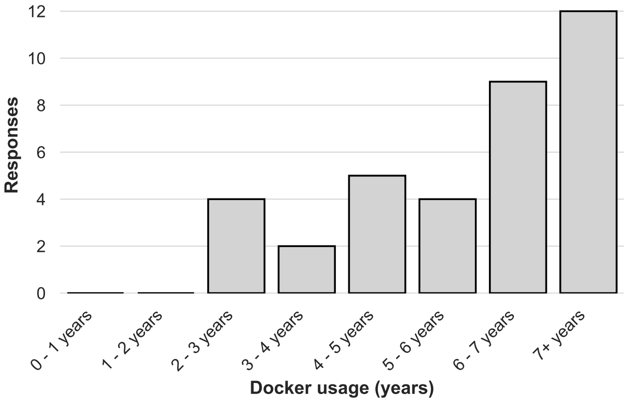

Figure 2: Docker experience (in years) of the survey participants.

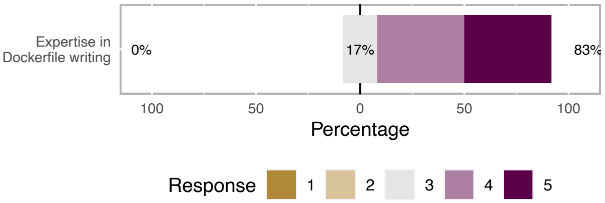

Figure 3: Dockerfile writing expertise measured using a Likert scale, varying from *1 - very inexperienced* to *5 - very experienced.*

Table 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		

sending the invitation, 7 of them answered the questions of the *post-questionnaire* within a period of ten days, approximately.

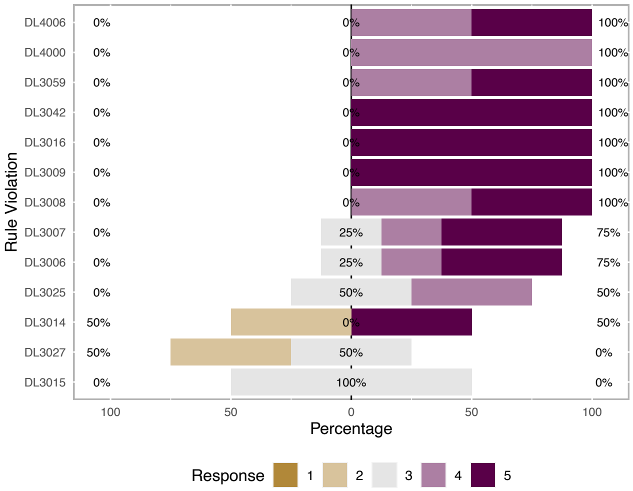

Figure 4: Developers' evaluation of the extent to which a Dockerfile smell should be avoided measured in S2Q1 (Table 2) using a 5-point Likert scale (from 1 - *"Strongly disagree"* to 5 - *"Strongly agree"*).

4.4.2 RQ$_2$: Are the Dockerfile smells considered bad practices by expert Dockerfile developers? In Table 3 we report the summary of the responses that we collected. 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 as a bad practice. Those are DL3003 (Use `WORKDIR` to switch 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 2).

In Fig. 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 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 or disagree*". None of the evaluations expresses 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 VSCode extension, *shellcheck* (some of the violations are contained also in *hadolint*), the *docker inspect* command, *dive*[13] (*i.e.,* a tool to inspect Docker image layers), and *shfmt*[14] (*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.

5 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).

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 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 conjecture that these Dockerfile smells depend on the context, and *hadolint* fails in catching this aspect.

For example, the following FROM instruction:

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

is identified as smell DL3006 (Missing version tag) by *hadolint*, which is not true. As reported by the *hadolint* community, in some cases knowing the context defines if a practice is "good" or "bad". An example is smell DL3020[15].

Another example is smell DL3008. In that case, pinning package versions requires continuous maintenance to keep versions up-to-date as it could lead to reliability issues in the future. This applies, specifically, to OS packages. Instead, missing version pinning for dependencies (*e.g.,* DL3013) is less diffused (Table 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). *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 (Avoid additional packages by specifying --no-install-recommends) are some examples that can help to reduce the image size.

Additionally, some of the unidentified violations are not harmful (*e.g.,* DL3003 - Use WORKDIR to switch directory). This also shows that not all the smells are a priority for the developers, specifically when there is not a direct impact on the resulting Docker image.

6 TOWARDS AN ENHANCED CATALOG OF DOCKERFILE SMELLS

The results of the two previous studies suggest 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 4), we found that they reported smells that are currently not part of the catalog captured by *hadolint* in 51% of the cases.

In this section, we wrap up all the obtained results to propose an enhanced catalog of smells, categorized in terms of their relevance to indicate what are the basic best practices that developers should care about when writing or maintaining Dockerfiles.

6.1 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 [33] to categorize the new recommendations provided by the participants in the open-ended questions. 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

[13] https://github.com/wagoodman/dive

[14] https://github.com/patrickvane/shfmt

[15] https://github.com/hadolint/hadolint/issues/693

which they have been reported. We report in Fig. 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 debug containers more easily. 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 best practices to follow to improve the build process of the Docker image. For example, copying and installing dependencies before sources (6 occurrences) allows to take advantage of the caching mechanism, speeding up the successive builds. Also, using a `.dockerignore` file (2 occurrences) allows to exclusion of unnecessary files from the build context to the image.

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.

Table 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 [20] (🔭), DRIVE [38] (🖌), and Dockercleaner [12] (🤖).

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

6.2 Ranking Dockerfile Smells

Many developers who have no or little expertise in writing Dockerfiles find themselves in need of writing or maintaining one. These developers do not know on what aspects to focus to have a good enough Dockerfile in terms of writing quality.

As a further contribution of this work, based on the frequency of the bad practices identified by the developers, we propose a *ranked* list of the Dockerfile smells analyzed in this work. 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 6.1). 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 4 we reported the ranked list of best practices. We also reported the overlap with other catalogs proposed in the literature, namely Binnacle [20], DRIVE [38], and DockerCleaner [12].

Thus, to meet a minimum quality level when writing Dockerfiles, developers should focus at least on the most frequently 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

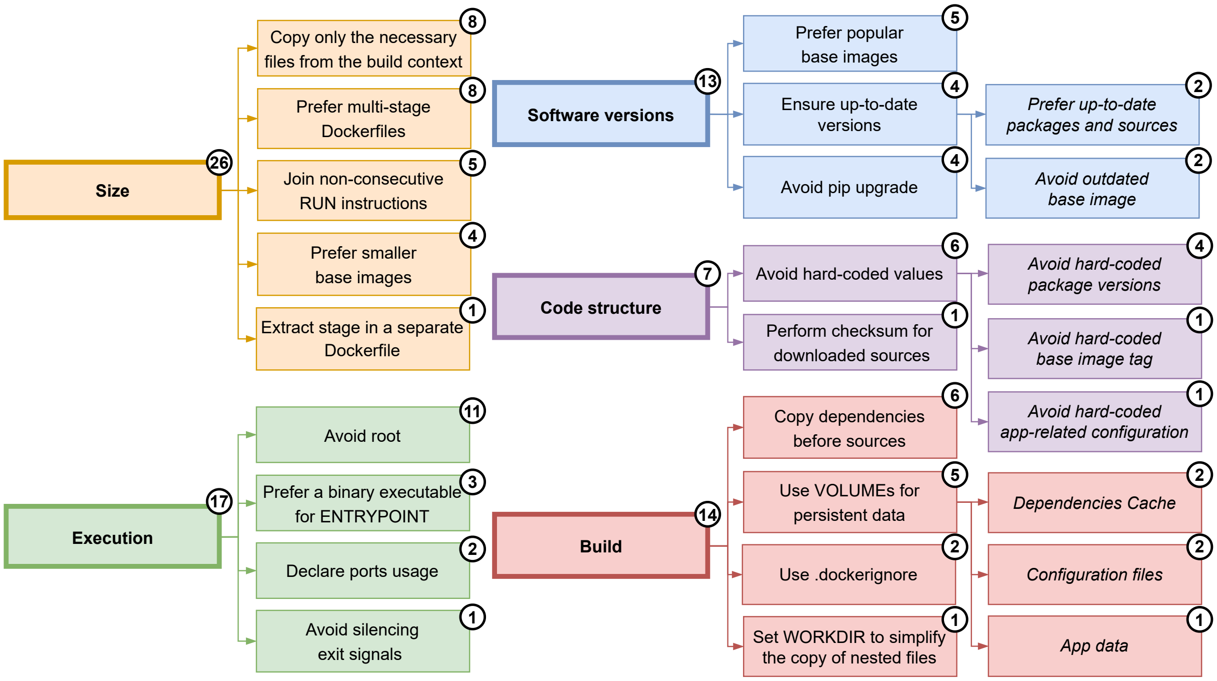
Size 26
Copy only the necessary files from the build context 8
Prefer multi-stage Dockerfiles 8
Join non-consecutive RUN instructions 5
Prefer smaller base images 4
Extract stage in a separate Dockerfile 1
Execution 17
Avoid root 11
Prefer a binary executable for ENTRYPOINT 3
Declare ports usage 2
Avoid silencing exit signals 1
Software versions 13
Prefer popular base images 5
Ensure up-to-date versions 4
Avoid pip upgrade 4
Prefer up-to-date packages and sources 2
Avoid outdated base image 2
Code structure 7
Avoid hard-coded values 6
Perform checksum for downloaded sources 1
Avoid hard-coded package versions 4
Avoid hard-coded base image tag 1
Avoid hard-coded app-related configuration 1
Build 14
Copy dependencies before sources 6
Use VOLUMEs for persistent data 5
Use .dockerignore 2
Set WORKDIR to simplify the copy of nested files 1
Dependencies Cache 2
Configuration files 2
App data 1

Figure 5: Categorization of the best practices recommendations provided by experts during our survey.

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[16]. 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.

7 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 in previous work. We found some customization 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 its 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 used in similar studies [39], 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.

8 CONCLUSION AND FUTURE WORK

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). Specifically, we focused on the writing practices captured by the state-of-the-practice *hadolint* tool. We first ran a study on official Dockerfiles to learn what smells appear most frequently in code written by *experts*. Then, we conducted a survey with expert Dockerfile developers to understand their perception of smells. We found that (i) official

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

Dockerfiles contain smells, and (ii) expert Dockerfile developers perceive some of the smells as more important than others. As a final result, we defined a prioritized catalog of smells that provides a clear guide to less experienced developers to write better Dockerfiles. We plan to further validate our prioritized catalog of smells through interviews with experts.

9 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 work by Gregorio Robles was supported by the project “Dependencias en colecciones complejas en módulos software” (PID2022-139551NB-I00), funded by the Spanish Ministry of Science, Innovation and Universities.

REFERENCES

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

[2] 2020. Contacting users for surveys. https://github.com/ghtorrent/ghtorrent.org/blob/master/faq.md#contacting-users-for-surveys. [Online; accessed 14-July-2023].

[3] 2020. Ethical issues to consider when conducting survey research. https://www.qualtrics.com/blog/ethical-issues-for-online-surveys/. [Online; accessed 14-July-2023].

[4] 2022. The all_packages pip package. https://pypi.org/project/all-packages/. [Online; accessed 28-July-2023].

[5] 2022. An introduction to the NPM package manager. https://nodejs.dev/en/learn/an-introduction-to-the-npm-package-manager/. [Online; accessed 28-July-2023].

[6] 2023. Best practices for writing Dockerfiles. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/. [Online; accessed 2-Jun-2022].

[7] 2023. CIS Docker benchmark. https://www.cisecurity.org/benchmark/docker. [Online; accessed 14-July-2023].

[8] 2023. Docker Bench for Security. https://github.com/docker/docker-bench-security. [Online; accessed 16-July-2023].

[9] 2023. Dockle - Container Image Linter for Security. https://github.com/goodwithtech/dockle. [Online; accessed 16-July-2023].

[10] 2023. OWASP Docker security cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/DockerSecurityCheatSheet.html. [Online; accessed 14-July-2023].

[11] 2023. Ruby gem stats. https://rubygems.org/stats. [Online; accessed 28-July-2023].

[12] Quang-Cuong Bui, Malte Laukötter, and Riccardo Scandariato. 2023. DockerCleaner: Automatic Repair of Security Smells in Dockerfiles. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, To Appear.

[13] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C Gall. 2017. An empirical analysis of the docker container ecosystem on github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 323–333.

[14] Kalvin Eng and Abram Hindle. 2021. Revisiting Dockerfiles in Open Source Software Over Time. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 449–459.

[15] Arlene Fink. 2003. *The survey handbook*. sage.

[16] Martin Fowler and Kent Beck. 1997. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*.

[17] Mubin Ul Haque, Leonardo Horn Iwaya, and M Ali Babar. 2020. 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)*. 1–11.

[18] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. 2023. Automating dependency updates in practice: An exploratory study on github dependabot. *IEEE Transactions on Software Engineering* (2023).

[19] Jordan Henkel, Christian Bird, Shuvendu K Lahiri, and Thomas Reps. 2020. A dataset of dockerfiles. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 528–532.

[20] Jordan Henkel, Christian Bird, Shuvendu K Lahiri, and Thomas Reps. 2020. Learning from, understanding, and supporting devops artifacts for docker. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 38–49.

[21] Yu Huang, Denae Ford, and Thomas Zimmermann. 2021. Leaving my fingerprints: Motivations and challenges of contributing to OSS for social good. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1020–1032.

[22] Barbara A Kitchenham and Shari L Pfleeger. 2008. Personal opinion surveys. *Guide to advanced empirical software engineering* (2008), 63–92.

[23] Emna Ksontini, Marouane Kessentini, Thiago do N Ferreira, and Foyzul Hassan. 2021. Refactorings and Technical Debt in Docker Projects: An Empirical Study. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 781–791.

[24] Jenny T Liang, Chenyang Yang, and Brad A Myers. 2023. Understanding the Usability of AI Programming Assistants. In *2024 IEEE/ACM 46rd International Conference on Software Engineering (ICSE)*. IEEE, To appear.

[25] Jenny T Liang, Thomas Zimmermann, and Denae Ford. 2022. 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*. 170–182.

[26] Changyuan Lin, Sarah Nadi, and Hamzeh Khazaei. 2020. 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)*. IEEE, 371–381.

[27] Vittoria Nardone, Biruk Muse, Mouna Abidi, Foutse Khomh, and Massimiliano Di Penta. 2023. Video game bad smells: What they are and how developers perceive them. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–35.

[28] Diogo Pina, Carolyn Seaman, and Alfredo Goldman. 2022. Technical debt prioritization: a developer’s perspective. In *Proceedings of the International Conference on Technical Debt*. 46–55.

[29] Giovanni Rosa, Simone Scalabrino, Gabriele Bavota, and Rocco Oliveto. 2023. What Quality Aspects Influence the Adoption of Docker Images? *ACM Transactions on Software Engineering and Methodology* (2023).

[30] Giovanni Rosa, Simone Scalabrino, Gregorio Robles, and Rocco Oliveto. 2024. *Replication package*. https://doi.org/10.6084/m9.figshare.23817024.v1.

[31] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2014. Measuring and modeling programming experience. *Empirical Software Engineering* 19 (2014), 1299–1334.

[32] Edward Smith, Robert Loftin, Emerson Murphy-Hill, Christian Bird, and Thomas Zimmermann. 2013. Improving developer participation rates in surveys. In *2013 6th International workshop on cooperative and human aspects of software engineering (CHASE)*. IEEE, 89–92.

[33] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.

[34] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. 2015. When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 403–414.

[35] Dirk Van Der Linden, Pauline Anthonysamy, Bashar Nuseibeh, Thein Than Tun, Marian Petre, Mark Levine, John Towse, and Awais Rashid. 2020. Schrödinger’s security: opening the box on app developers’ security rationale. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 149–160.

[36] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. 2020. Characterizing the occurrence of dockerfile smells in open-source software: An empirical study. *IEEE Access* 8 (2020), 34127–34139.

[37] Xiaoya Xia, Shengyu Zhao, Xinran Zhang, Zehua Lou, Wei Wang, and Fenglin Bi. 2023. Understanding the Archived Projects on GitHub. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 13–24.

[38] Yu Zhou, Weilin Zhan, Zi Li, Tingting Han, Taolue Chen, and Harald Gall. 2022. DRIVE: Dockerfile Rule Mining and Violation Detection. *arXiv preprint arXiv:2212.05648* (2022).

[39] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2019. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering* 47, 10 (2019), 2084–2106.