Empirical Study of the Docker Smells Impact on the Image Size

Durieux, Thomas

DOI
10.1145/3597503.3639143

Publication date
2024

Document Version
Final published version

Published in
Proceedings - 2024 ACM/IEEE 44th International Conference on Software Engineering, ICSE 2024

Citation (APA)
Durieux, T. (2024). Empirical Study of the Docker Smells Impact on the Image Size. In *Proceedings - 2024 ACM/IEEE 44th International Conference on Software Engineering, ICSE 2024* (pp. 2568-2579). (Proceedings - International Conference on Software Engineering). IEEE. https://doi.org/10.1145/3597503.3639143

Empirical Study of the Docker Smells Impact on the Image Size

Thomas Durieux
TU Delft
The Netherlands
thomas@durieux.me

ABSTRACT

Docker, a widely adopted tool for packaging and deploying applications leverages Dockerfiles to build images. However, creating an optimal Dockerfile can be challenging, often leading to "Docker smells" or deviations from best practices. This paper presents a study of the impact of 14 Docker smells on the size of Docker images.

To assess the size impact of Docker smells, we identified and repaired 16 145 Docker smells from 11 313 open-source Dockerfiles. We observe that the smells result in an average increase of 48.06 MB (4.6 %) per smelly image. Depending on the smell type, the size increase can be up to 10 %, and for some specific cases, the smells can represent 89 % of the image size. Interestingly, the most impactful smells are related to package managers which are commonly encountered and are relatively easy to fix.

To collect the perspective of the developers regarding the size impact of the Docker smells, we submitted 34 pull requests that repair the smells and we reported their impact on the Docker image to the developers. 26/34 (76.5 %) of the pull requests have been merged and they contribute to a saving of 3.46 GB (16.4 %). The developer's comments demonstrate a positive interest in addressing those Docker smells even when the pull requests have been rejected.

CCS CONCEPTS

• **Software and its engineering** → *Software evolution*; **Maintaining software**.

ACM Reference Format:
Thomas Durieux. 2024. Empirical Study of the Docker Smells Impact on the Image Size. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639143

1 INTRODUCTION

Docker is a widely adopted tool among developers and organizations for packaging, deploying, and running applications in lightweight, portable containers. A critical component of Docker is the Dockerfile, a straightforward text file based on shell that outlines the necessary steps to build a Docker image. However, creating an optimal Dockerfile can be challenging, particularly when shell best practices differ from the ones in Docker. When there is a deviation from these best practices, we refer to it as a "Docker smell". Docker smells are commonly found within Dockerfiles because many developers who create them may lack expertise in this area [15]. Furthermore, the best practices used in interactive shells often contrast with those applicable to shells within Dockerfiles, resulting in suboptimal Docker images.

Previous research conducted by academics and industry has primarily focused on detecting Docker smells. Several linters, such as *Binnacle* [15], *hadolint* [13], *dockerfilelint* [6], *docker-bench-security* [5], and *dockle* [7], have been developed specifically to identify a wide range of Docker smells. However, these tools suffer from limited recognition in the developer community as multiple studies show the almost systematic presence of smells in Dockerfiles [3, 21]. One possible reason for the lack of recognition among developers may be the absence of studies on the impact of these smells, making it challenging to justify investing effort into addressing them.

In this contribution, we aim to address this specific problem by investigating the impact of Docker smells on the image size. We focus on the image size since it impacts multiple aspects of the Docker ecosystem. Firstly, the image size impacts the Docker images selection by the developers, smaller images have more chances to be selected [27]. It also contributes to the size of the Docker registry which was already reaching 1 PB in 2019 [37] for public repositories and is expected to be much bigger for private repositories. It also impacts the download latency of the Docker images [35] which is problematic in large deployment environments, as well as increases the attack surface of the Docker images.

In this study, we investigate the size impact of 14 Docker smell types originally identified by Henkel et al. [15] as having a potential impact on image size. The size impact is measured by identifying and removing 16 145 real Docker smells from 11 313 open-source Dockerfiles. We then investigate the developers' perspectives and interests in those Docker smells in order to identify if notifying the developers about those smells is relevant or not. This aspect has been performed by opening 34 pull requests that repair and report the impact of 78 Docker smells. The detection and repair are performed by our tool, `Parfum`, which has been specifically developed for this purpose.

Our observations reveal that Docker smells exert a substantial impact on the size of Docker images. On average, the Docker smells lead to a size increase of 48.06 MB or 4.6 % per image. Additionally, this bloat translates to a total additional 2.05 TB in transferred data per week on DockerHub. Notably, we found that the most impactful smells identified in this study are associated with the utilization of package manager commands. Those smells also happen to be among the most frequently encountered ones, which means that identifying and repairing a few smells can have a huge impact and improve the quality of the Docker images.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639143

As direct evidence of the relevance of smells repair, 26/34 (76.5 %) of pull requests have been successfully merged, indicating developers' interest in repairing Docker smells, 6 pull requests are still waiting for an answer and 2 pull requests have been rejected because the proposed changes were already included in the repository. The merged pull requests contribute to a saving of 3.46 GB (16.4 %).

In summary, the contributions of this paper include:

- An empirical study on the impact of Docker smells on the image size,
- A new dataset of 159 748 Dockerfiles extracted from GitHub and a ground truth dataset of 384 Dockerfiles,
- 34 pull requests that repair 78 Dockerfile smells,
- `Parfum`, a tool that detects and repairs automatically 14 types of Docker smells.

We are pleased to announce that we have made the results of our study accessible at [8]. Additionally, the smell detection and repair technique is available at [9] and can be tested at https://durieux.me/docker-parfum.

2 BACKGROUND

In this section, we provide the key concepts and background information required for our study.

Containers are a form of virtualization technology designed to offer a more efficient and streamlined approach to software deployment. Unlike traditional virtual machines, containers encapsulate applications and their dependencies, ensuring consistency across different environments. By doing so, they enhance portability and facilitate the seamless movement of applications between development, testing, and production environments. Containers gained popularity also due to their lower overhead compared to virtual machines [1, 19].

Docker is the most popular container platform that can create, deploy, and run containerized applications.[1] Docker also has its own Docker registry which is the most popular registry for open-source Docker images.

Docker Image is an executable package for Docker that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and config files. Docker images are built using instructions contained in a Dockerfile.

Dockerfile is a text file that contains instructions for building a Docker image. The instructions define the base image to use (`FROM <image>`), the files to include (`COPY <source> <dest>`), the ports to open (`PORT <port>`), the entry point (`ENTRYPOINT <script>`), and the scripts to execute (`RUN <script>`). The scripts declared in the Dockerfiles define the actions that need to be performed to create the Docker image. Those scripts are shell commands which are generally bash or PowerShell (for Windows Docker image).

Docker Smell refers to a potential issue, problem, or suboptimal configuration with a Dockerfile or Docker image [31]. This issue is generally detected when the Dockerfile or image violates some best practices. Common Docker smells include bloated images, misconfiguration, misuse of commands, and security issues. Identifying and addressing these smells can help improve the efficiency, security, and maintainability of a Docker-based project [31]. In this paper, we focus on smells inside the Dockerfiles.

Binnacle by Henkel et al. [15] is a tool that studies and detects Docker smells in Dockerfiles. The particularity of this work compared to other linters such as Hadolint is that it not only detects Docker smells but also analyzes the presence of those smells inside GitHub and compares it to a high-quality set of Dockerfiles. Additionally, it also categorizes the impacts of the smells they observed. Interestingly, the majority of the smells are related to space waste; this observation initiated this study to measure the actual impact of those smells on the image size. The Docker smells reported by Binnacle have a small overlap with other existing linters such as Hadolint which only supports 4 smell types, also supported by Binnacle, that impact image size. Indeed, most of the Binnacle smells are related to the shell while Hadolint focused on the Docker instructions and the size impact is related to the shell usage.

3 METHODOLOGY

We describe the empirical study we conduct on the impact of Docker smells. We first present the methodology that we follow to perform this empirical study. Then, we present the datasets that we use for the empirical study. We follow by describing how we detect and repair Docker smells with our tool called `Parfum`.

3.1 Methodology Overview

In order to measure the impact of the smells on image size. We follow the following methodology for each Dockerfile. First, we identify the smells that are present inside the Dockerfiles. If a smell is present, we build the smelly Dockerfile to produce a Docker image and we measure the size of that image. We then repair the smell and produce a new Dockerfile without smells. We then build the repaired Dockerfile and measure the size of the new image. Finally, the difference in size between the original and the repaired image is the impact of the detected smells.

3.2 Research Questions

In this section, we present the impact of Docker smells on Docker image size. We design and conduct an empirical evaluation to answer the following research questions:

RQ1 **What is the effectiveness of our approach in detecting and repairing Docker smells?** This first research question aims to validate a crucial aspect of our methodology: being able to detect and repair Docker smells. To do so, we first measure the effectiveness of smell detection on a ground truth dataset. Then, we conduct a quantitative analysis of the repair of 164 597 Dockerfiles that contain at least one smell. Finally, we selected 11 313 smelly Dockerfiles and built them to ensure that the repairs do not break the Docker builds.

RQ2 **What is the impact of the identified Docker smells on the Docker image size?** In this second research question, we study the impact of the Docker smell on the size of the Docker images. To answer this question, we measure the image size before and after the repair for the 4827 Dockerfiles. We also study which smells have the most impact the most the Docker image size and the effect of the smells on the DockerHub bandwidth. Finally, we measure the impact of the smells in terms of bandwidth on Dockerhub.

[1] Docker: https://www.docker.com

RQ3 **What is the developers' attitude towards Docker smells?** In the final research question, we aim to evaluate the interest that the developers have in the repair of Docker smells impacting image size. To do so, we opened 34 pull requests that fix the identified smells and we analyzed the responses of the developers.

By addressing these research questions, we aim to analyze the Docker smells impact on image size and developers' attitudes regarding these smells.

3.3 Docker Smells

As previously mentioned, for this study we focus on the Docker smell that introduces an increase in size as presented by Henkel et al. [15]. We therefore ignore the smells that are related to the security or build reliability. During this study, we will therefore focus on 14 smells. Table 1 describes the smells and provides the ID that we will use to refer to them. Additionally, the table includes the results of our first research question which we will present later on.

3.4 Datasets

In order to study the impact of the smells we had to select and create a new dataset of Dockerfiles. This section will present the dataset that we use in this study. Table 2 gives an overview of the main characteristics of our datasets and highlights some of their main differences.

3.4.1 Ground Truth Dataset. The second dataset that we consider in this study is a dataset of 384 unique Dockerfiles. That is used to measure the effectiveness of our approach to detect Docker Smells. The Dockerfiles have been manually annotated to identify the Docker smells. We randomly selected those Dockerfiles from the Binnacle Dataset. We chose a sample size of 384 Dockerfiles to obtain a dataset that is representative of the Binnacle dataset with a confidence level of 95 % and with a margin of error of 5 % according to the Cochran's Sample Size Formula: $n = \frac{z^2 \cdot p(1-p)}{\epsilon^2}$, where n is the required sample size, z is the Z-score corresponding to the desired confidence level (e.g., 1.96 for a 95 % confidence level), p is the estimated proportion of the population with a certain characteristic, and ϵ is the desired margin of error [4]. The ground truth dataset is also available on our online artifact [8]. We observe that the distribution of the number of instructions in the ground truth dataset and the Binnacle dataset are similar and therefore we are confident that this annotated dataset is representative.

The methodology to create this dataset is as follows: 1. The authors read the description of the smells to have a clear understanding of the smells. 2. We create a dashboard that displays and annotates the Dockerfiles; the goal is to minimize the effort of the annotation and to focus on the manual detection of the smells. 3. Multiple interactions have been performed to ensure that all smells are identified. 4. As a final check, we carefully analyze the results of Binnacle and ours to identify cases that could have been mislabeled.

At the end of this process, 152 Dockerfiles have at least one smell, and in total 468 smells have been annotated. This dataset is as far as we know the first ground truth dataset for Docker smells.

3.4.2 Binnacle Dataset. The first dataset that we use is the dataset of unique Dockerfiles presented in the Binnacle paper [15], which contains 178 452 Dockerfiles extracted from GitHub repositories in 2020. The main purpose of this dataset is to compare our smell detection to the baseline: Binnacle.

3.4.3 `Parfum` Dataset. The third and final dataset contains 159 748 Dockerfiles that were extracted from GitHub repositories in 2022. This dataset is used to study the impact of the smells in RQ2 and identify projects where we submit pull requests in RQ3. We could not use the Binnacle dataset for RQ2 and RQ3 because we could not identify from which repository the Dockerfiles from the Binnacle dataset were and therefore we could not build the Docker images to measure their size nor open pull requests. To avoid this problem in the future, we include in our dataset the origin repository, commit SHA, and the path of the Dockerfile.

The methodology for creating this new dataset is described in the following. The first step is to identify an initial set of GitHub repositories. We decided to select repositories that are 1) not forks, 2) have at least 10 stars, and 3) have at least 50 commits. We choose those criteria to obtain Dockerfiles from repositories that have a minimum of activity and that are more likely to have been maintained. We ended up with a list of 500 108 potential repositories.[2]

The next step is to download the file list from the default branch of the latest commit for each repository. We were able to download the file list for 500 022 repositories, the missing file lists are due to unreachable repositories.

The following step is to identify and download the Dockerfiles stored in these repositories. We iterated over the list of files and considered any files that contained the string "Dockerfile" (case sensitive) as potential Dockerfiles. Finally, we identified the unique Dockerfiles that we use in this study. This resulted in a collection of 159 748 Dockerfiles that constitute the new dataset which is available on our online artifact [8] as well as the scripts that are used to generate the dataset.

3.5 `Parfum`

In this section, we present, `Parfum`, a tool we use to detect and repair Docker smells. `Parfum` is available on GitHub [9] and it also has been ported to a browser version which is available at https://durieux.me/docker-parfum.

`Parfum` detection of smells is inspected by Binnacle [15] and supports the smells that Binnacle reports as being related to space waste. The major difference between Binnacle and `Parfum` is that `Parfum` repairs those smells and it also links the smells to an AST node which allows much more precise analysis and extensions.

3.5.1 `Parfum` Steps. In this section, we briefly explain how `Parfum` works by presenting the six main steps.

1. **Parsing Dockerfile AST**: The first step of `Parfum` is to parse the Abstract Syntax Tree (AST) representation of the Dockerfile.
2. **Parsing shell commands**: `Parfum` parses each Docker command that includes a shell command, i.e., `RUN <cmd>` and compiles it with the Dockerfile AST to form a unified AST.

[2] Downloaded on July 12, 2022 from https://seart-ghs.si.usi.ch/

Table 1: The considered Docker smells and the detection rate by `Parfum` and Binncale in our Ground Truth dataset.

#	Smell ID	Smell Description	`Parfum`	Binnacle
1	pipUseCacheDir	Clean cache after `pip install`.	82/82 (100.0 %)	67/82 (81.7 %)
2	npmCacheCleanUseForce	Clean cache after `npm install`.	2/2 (100.0 %)	2/2 (100.0 %)
3	mkdirUsrSrcThenRemove	Remove /usr/src/* after usage.	-	-
4	rmRecurisveAfterMktempD	Remove temporary folders.	-	-
5	tarSomethingRmTheSomething	Remove tar files after decompression.	13/12 (108.3 %)	7/12 (58.3 %)
6	apkAddUseNoCache	Use `--no-cache` flag with `apk add`.	8/8 (100.0 %)	8/8 (100.0 %)
7	aptGetInstallUseNoRec	Use `--no-install-recommends` flag in `apt-get install`.	159/159 (100.0 %)	122/159 (76.7 %)
8	aptGetInstallRmAptLists	Remove `/var/lib/apt/lists/*` after `apt-get install`.	153/153 (100.0 %)	117/117 (100.0 %)
9	gpgVerifyAscRmAsc	Remove .asc file after usage.	-	-
10	npmCacheCleanAfterInstall	Force to clean cache after `npm install`.	30/30 (100.0 %)	28/28 (100.0 %)
11	gemUpdateSystemRmRootGem	Clean cache after `gem update --system`.	1/1 (100.0 %)	1/1 (100.0 %)
12	gemUpdateNoDocument	Add `--no-document` flag to the .gemrc config file.	1/1 (100.0 %)	1/1 (100.0 %)
13	yumInstallRmVarCacheYum	Clean cache after `yum install`.	17/17 (100.0 %)	17/17 (100.0 %)
14	yarnCacheCleanAfterInstall	Clean cache after `yarn install`.	3/3 (100.0 %)	0/3 (0.0 %)

Table 2: Characteristics of Binnacle [15], `Parfum`, and Ground Truth datasets.

Metric	Binnacle	`Parfum`	Ground Truth
Creation date	2020	July 2022	July 2022
# Dockerfile	178 452	159 748	384
# Smelly Dockerfile	72 313	89 143	152
Total # Instruction	2 223 139	3 637 952	4938
Avg. # Instruction	12.45	18.04	12.86
Med. # Instruction	9	12	9

(3) **Enriching the Docker AST**: Next, `Parfum` enriches the Docker AST by incorporating structural information from the command lines. For instance, consider the command `RUN apt-get install wget` and its AST representation. The enriched AST contains annotations specifying that `apt-get` is used to `install` packages, and the installed package is `wget`. These annotations are added to the corresponding nodes in the AST, highlighting their roles and relationships and they can be used later on by the smell analyzer. `Parfum` supports a total of 88 command lines, which account for 89.05 % of all the commands found in the Dockerfiles within our dataset. The remaining commands consist of either custom or infrequent commands. Consequently, these commands will not be part of Docker smells by nature.

(4) **Enriching embedded commands**: `Parfum` enriches commands that are embedded within other commands. For example, the command `sudo apt update` contains a main command (`sudo`) and an embedded command (`apt update`).

(5) **Detecting Docker smells**: The detection of smells is made by querying the AST. Each smell is associated with an AST query. The detection of smells is detailed in Section 3.5.2.

(6) **Repairing Docker smells**: Once Docker smells are detected, `Parfum` can proceed with the repair. We employ a template-based approach to repair the smells, the details of the repair are available in Section 3.5.3.

3.5.2 Smell Detection. The smell detection of `Parfum` uses a template matching system to identify patterns inside the Dockerfile AST. In total, `Parfum` supports 32 Docker smell detections, but we only considered the 14 that are related to space waste. The list of the 32 supported smells is presented in our repository [9], even if they are not the focus of this paper, developers can still use `Parfum` to detect and fix them.

The considered smells are described in Section 3.3. Each rule is defined as a query, specifying the required AST nodes that need to be present to trigger the smell. An additional post-condition specifies additional AST nodes that should be present before, after, or inside the matched node. Figure 1a presents an example of such a template matching. In this example, we detect that the flag `-f` is missing within the command `npm cache clean`. In this example, we look for the command `npm cache clean` using the query `Q("NPM-CACHE-CLEAN")`. The post-condition verifies that the flag `-f` is not present inside the node with the query `Q("NPM-F-FORCE")`. If those two queries have a match, `Parfum` has detected the smell and it is reported to the developer.

3.5.3 Smell Repair. Once a smell is detected, `Parfum` repairs the Dockerfile by modifying its AST. This is a novelty of `Parfum`, as far as we know `Parfum` is the first tool that fixes smells in Dockerfiles. Figure 1b presents an example of how the `Parfum` modifies the AST to fix the smell. In this particular example, the smell is related to the command `npm cache clean`. `Parfum` repairs the smell by adding the `--force` flag as an argument to the `npm cache clean` command. Once the AST is transformed, the detection of the smell is triggered again to verify that the repair was made properly. If the smell is still detected, the repair is rollback to avoid introducing inappropriate changes to the Dockerfiles.

```
{
 // look for `npm cache clean`
 query: Q("NPM-CACHE-CLEAN"),
 consequent: {
  // look for `--force` flag
  inNode: Q("NPM-F-FORCE") }
}
```

(a) Detect smell.

```
function repair(node) {
  // insert --force flag
  node.addChild(BashCommandArgs().addChild(
    BashLiteral("--force")
  ));
}
```

(b) Repair smell.

```
@@ -21,1 +21,1 @@
-RUN npm cache clean
+RUN npm cache clean --force
```

(c) Generated Dockerfile patch.

After modifying the AST, Parfum can reprint the AST into a Dockerfile. The reprinting process in Parfum utilizes a pretty-print feature, resulting in the reprinted AST containing only the modified nodes. This approach minimizes the changes made to the Dockerfile while addressing the detected smells. An example of such transformation can be seen in Figure 1c, showcasing the differences between the original Dockerfile and the repaired version.

4 STUDY RESULTS

In this section, we present and discuss the answers to our research questions.

4.1 RQ1: Smell Detection & Repair Effectiveness

In this first research question, we assess the effectiveness of our methodology in identifying and repairing Docker smells. The detection and repair are handled by our tool: Parfum and consequently, we will also evaluate the effectiveness of our tool. To simplify the narrative, we will refer to our approach as Parfum in this research question.

This evaluation is divided into two parts: first, we evaluate the effectiveness of detecting the smells by analyzing the smell detection rate of our approach on the ground truth dataset and also comparing it to the baseline: Binnacle [15]. Second, we evaluate the repair effectiveness of our approach and its impact on the Docker build, i.e., build failure rate.

4.1.1 Parfum vs Binnacle. To increase our confidence in our approach, we measure our approach detection rate and compare it to the baseline, Binnacle, on our ground truth dataset (see Section 3.4.1). Table 1 presents the detection rate of Parfum and Binnacle. We observe that Parfum has almost a perfect detection rate. Only in one case, Parfum produces a false positive for the smell *tarSomethingRmTheSomething* while Binnacle produces at least 89 false negatives. We cannot get the precise rate because Binnacle only reports the number of each detected smell without their position which could lead to an unprecise comparison with the ground truth.

Listing 1 presents the only false positive reported by Parfum. It happens because Parfum does not succeed in identifying that the developers already removed the tar using the command: `rm -rf /tmp/firefox.*`.

Based on those results, we can be confident in the detection effectiveness of our approach.

4.1.2 Parfum Repair Effectiveness. We are now looking at the effectiveness of Parfum to repair the Docker smells. As far as we know no tool or dataset could be used to compare the results of Parfum.

```
RUN FIREFOX_URL="https://download.mozilla.org/?pro⌋
↪  duct=firefox-latest-ssl&os=linux64&lang=en-US"
  && ACTUAL_URL=$(curl -Ls -o /dev/null -w
  ↪  %{url_effective} $FIREFOX_URL)
  && curl --silent --show-error --location --fail
  ↪  --retry 3 --output /tmp/firefox.tar.bz2
  ↪  $ACTUAL_URL
  && sudo tar -xvjf /tmp/firefox.tar.bz2 -C /opt
  && sudo ln -s /opt/firefox/firefox
  ↪  /usr/local/bin/firefox
  && sudo apt-get install -y libgtk3.0-cil-dev
  ↪  libasound2 libasound2 libdbus-glib-1-2
  ↪  libdbus-1-3
  && rm -rf /tmp/firefox.*
  && firefox --version
```

Listing 1: False positive produced by Parfum. Parfum did not identify that firefox.tar.bz2 was removed by rm -rf /tmp/firefox.* commands and therefore identifies the *tarSomethingRmTheSomething* smell in this snippet.

Therefore, we use Parfum to automatically repair the smells in the $178463(\textit{Binnacle Dataset}) + 159748(\textit{Parfum Dataset}) = 338211$ Dockerfiles, 164 597 (48.7 %) of them contain at least one smell. We then verified that the smells were fixed by analyzing the repaired Dockerfiles. Due to the high detection rate presented in the first part of this research question, we can be confident about the repair rate. To increase our confidence in checking if Parfum is not breaking builds, we built the Docker image for a selection of Dockerfiles to ensure that the repair did not break the Docker build. This will give us some indications of the reliability of the repair. We do know that it is not a perfect oracle and it does not guarantee that the behavior of the images is preserved. However, we could not identify a way that would allow us to verify the behavior of Docker images at a meaningful scale.

Table 3 presents the results of the smell repairs. The first column of Table 3 contains the name of the smell, and the second column contains the number of occurrences of this smell. The third column contains this information after the repair. The fourth and fifth columns contain the same information but instead count the number of Dockerfiles, i.e., a Dockerfile can contain more than one occurrence of a specific smell. The results show that Parfum is able to repair 514 010 (99.8 %) Docker smells. The smell *aptGetInstallThenRemoveAptLists* is the smell that is the most present after repair

Table 3: The occurrence of each smell before and after the repair using Parfum on Binnacle and Parfum datasets.

Docker Smell	# Docker Smell		# Dockerfile with Smell	
	Before Repair	After Repaired	Before Repair	After Repaired
pipUseNoCacheDir	76 856 (14.9 %)	7 (0.9 %)	41 282 (25.1 %)	3 (0.6 %)
npmCacheCleanUseForce	2447 (0.5 %)	6 (0.8 %)	2413 (1.5 %)	6 (1.2 %)
mkdirUsrSrcThenRemove	4777 (0.9 %)	28 (3.6 %)	4329 (2.6 %)	27 (5.5 %)
rmRecursiveAfterMktempD	768 (0.1 %)	11 (1.4 %)	491 (0.3 %)	11 (2.2 %)
tarSomethingRmTheSomething	20 902 (4.1 %)	129 (16.5 %)	14 660 (8.9 %)	97 (19.7 %)
apkAddUseNoCache	15 094 (2.9 %)	0 (0.0 %)	11 671 (7.1 %)	0 (0.0 %)
aptGetInstallUseNoRec	172 028 (33.4 %)	1 (0.1 %)	81 448 (49.5 %)	1 (0.2 %)
aptGetInstallThenRemoveAptLists	142 187 (27.6 %)	389 (49.7 %)	74 958 (45.5 %)	209 (42.5 %)
gpgVerifyAscRmAsc	157 (0.0 %)	0 (0.0 %)	144 (0.1 %)	0 (0.0 %)
npmCacheCleanAfterInstall	32 437 (6.3 %)	63 (8.0 %)	24 248 (14.7 %)	52 (10.6 %)
gemUpdateSystemRmRootGem	505 (0.1 %)	2 (0.3 %)	457 (0.3 %)	2 (0.4 %)
gemUpdateNoDocument	390 (0.1 %)	0 (0.0 %)	345 (0.2 %)	0 (0.0 %)
yumInstallRmVarCacheYum	24 124 (4.7 %)	95 (12.1 %)	12 083 (7.3 %)	53 (10.8 %)
yarnCacheCleanAfterInstall	5010 (1.0 %)	12 (1.5 %)	4041 (2.5 %)	12 (2.4 %)
Total	514 793	783	164 597	492

Table 4: The number of build errors per smell.

Docker Smell	# Build Errors
aptGetInstallUseNoRec	312 (84.6 %)
aptGetInstallThenRemoveAptLists	254 (68.8 %)
pipUseNoCacheDir	115 (31.2 %)
npmCacheCleanAfterInstall	32 (8.7 %)
tarSomethingRmTheSomething	48 (13.0 %)
apkAddUseNoCache	17 (4.6 %)
mkdirUsrSrcThenRemove	6 (1.6 %)
yumInstallRmVarCacheYum	4 (1.1 %)
gemUpdateSystemRmRootGem	1 (0.3 %)
gemUpdateNoDocument	1 (0.3 %)
npmCacheCleanUseForce	1 (0.3 %)

followed by *tarSomethingRmTheSomething* and *yumInstallRmVarCacheYum*. However, those cases are rare and should not impact significantly the results of our study.

We now verify that Parfum does not break the build. To do so, we build the Docker images before and after the repair. We could not scale the build of the 164 597 Dockerfiles due to the amount of computing it would have required. Indeed, it takes on average 8m 37s to build a Docker image. Additionally, the rate-limited imposed by Dockerhub would also block to perform this experiment on all the images. Instead, we selected all the Dockerfiles that are located at the root of the repositories, that are exactly named `Dockerfile`, and that contain at least one smell. We chose those criteria because we expect that those Dockerfiles are the main Dockerfiles of the repositories. We end up with 11 313 Dockerfiles and we only succeeded in building 5196 (45.9 %) of them which illustrates the complexity of reproducing Docker builds.

Once, we identify the 5196 Dockerfiles that are buildable and apply Parfum on them, and proceed to rebuild the Dockerfiles after the repair. 4827 Dockerfiles build after the repair which results in 369 (7.1 %) build failures (or build flakiness). It is difficult to estimate the number of builds that are failing due to the build flakiness. However, it is reasonable to believe that the majority is due to Parfum repairs.

```
RUN wget -O gsl.tgz ftp://ftp.gnu.org/gsl-1.16.tar
  && tar -zxf gsl.tgz && mkdir gsl
  && cd gsl-1.16 && ./configure --prefix=/app/gsl
  && make && make install
  && rm gsl.tgz                      # Added line
```

Listing 2: Example of invalid repair made by Parfum for the repository github.com/olavolav/te-causality.

Table 4 presents the number of builds that finish with an error per smell type. Note that we consider that all applied repairs have impacted the build status. We observe that the vast majority of the errors are related to the rules *aptGetInstallUseNoRec* and *aptGetInstallThenRemoveAptLists*. Those rules can break builds when a recommended package is removed when it is required or when Parfum removes the cache when it was already empty.

In a few cases, Parfum produces invalid repairs such as Listing 2. In this case, Parfum places `rm gsl.tgz` after the change of directory (`cd gsl-1.16`), the file `gsl.tgz` is, therefore, not found and the build fails.

Answer to RQ1. We show that our approach is able to detect all the Docker smells in our ground truth dataset with only one false positive while also being able to repair 99.8 % of the smells. We broke 7.1 % of the builds, but it is acceptable for developers that are able to tolerate from 15 % to 20 % of false positives that developers would tolerate [2]. We conclude that our approach is suitable for measuring the size impact of the smells. By side effect, we also show that Parfum is effective and could be used by practitioners to detect and fix Docker smells.

4.2 RQ2: Impact of Docker Smells

In this research question, we investigate the impact of Docker smells on the size of Docker images. We utilize the 5196 buildable Dockerfiles from the previous research question and analyze the differences between the images before and after the repairs. Our investigation focuses on image size impact, and on bandwidth usage introduced by the smells.

The results of the size impact investigation are presented in Table 5. The table lists the names of the smells, along with the space used by each smell (difference before and after repair), average used space, median used space, and maximum used space.

It is crucial to acknowledge that while we can observe the space savings per Dockerfile, it is not possible to determine the exact space savings for each smell since we built the Docker images once with all repairs applied.

Overall, the identified smells contribute to an increase in the image size of 277.03 GB (approximately 4.66 %). On average, each Dockerfile exhibits an increase of approximately 48.06 MB in terms of size, with a median of 1.9 MB per Dockerfile.

We performed the Wilcoxon signed-rank test to verify if the reduction of size is a significant difference. Wilcoxon signed-rank test is used to compare the locations of two populations using two matched samples and this test is also compatible with non-normal data as is the case here as observed by the Shapiro normality test. We consider that the reduction in size is significant if the $p-value$ is lower than 0.05. We obtained a $p-value$ value of 0 which indicates a significant difference in the size before and after the repair.

We also observe a variation in terms of size impact depending on the smell. Some smells, like *npmCacheCleanUseForce*, result in an average impact of approximately 10 %. While other smells like *mkdirUsrSrcThenRemove* only have an impact of 1.1 %. In general, the smells that primarily impact image size are related to package managers, particularly instances where developers forget to remove caches, such as *aptGetInstallThenRemoveAptLists*, *pipUseNoCacheDir*, *npmCacheCleanAfterInstall*, and *aptGetInstallUseNoRec*. These smells are not only among the most frequent but are also relatively straightforward to address.

Additionally, considering the number of times Docker images are downloaded from DockerHub, the impact of these smells becomes more significant. Table 6 presents the impact of the smells on the bandwidth of DockerHub. This table only considers the 1511 Docker images that we found on DockerHub. [3] We estimate that the detected smells result in an increase of 40.45 TB of data transfer per week on DockerHub. This estimation considers the total downloads for each Docker image and the size difference between the original and repaired Docker images, divided by the median image compression ratio (3.2x) reported by Zhao et al. [36].

Those numbers can seem non-meaningful for a company the size of DockerHub. However, we measured the impact on a small number of images, Dockerhub contains at least used 636 625 unique images [24] that are pulled 446 billion times. While considering the full scale of DockerHub those smells have a measurable impact on DockerHub.

```
Hi there,

I've made a small improvement to the Dockerfile
 ↪  that I think could help optimize the image
 ↪  size.

Summary of the changes:
- <change description>

Impact on the image size:

Image size before repair: <size> MB
Image size after repair: <size> MB
Difference: <size> MB
I hope that you will find these changes useful to
 ↪  you. Let me know if you have any questions or
 ↪  concerns.

Thanks,
```

Listing 3: Template of the pull request description that we used to propose Docker smell repair.

Answer to RQ2. Docker smells significantly impact the size of Docker images, with an average of 4.66 % and going up to 10 % for some of the smells. This leads to an additional 2.05 TB of data transfer per week on DockerHub for 1511Docker images. Among the most frequent and impactful smells we identified, many are related to the use of package managers and their caches. Addressing these smells can have a substantial effect on image size and overall image efficiency.

4.3 RQ3: Developers' Attitude Towards Docker Smells

In this final research question, we investigate developers' attitudes toward Docker smells and their impact. The main goal is to validate the relevance of these smells to developers and the importance of addressing them. To gather feedback from developers, we opened pull requests that addressed Docker smells, and we assessed the impact of these smells within the pull request descriptions. A template of this description is available in Listing 3.

We established the following criteria to select the repositories where we would submit the pull requests: (1) Dockerfile has at least one smell and less than ten. (2) The repository is active (not archived, not a fork, has open issues, at least one fork, and has a commit in the last two months preceding the date of the study on the main branch). (3) The Docker image builds successfully after the repair. (4) No more than one pull request per GitHub organization. (5) The Docker image has been downloaded at least 1000 times from Dockerhub. (6) The size difference needs to be larger than 1 Mb. Following these criteria, we identified 124 potential candidates and selected 34 repositories that explicitly welcome external contributions. The list of opened pull requests can be found in our repository [8].

The merged and closed pull requests are presented in Table 7. The table includes the repository name, the number of stars, and the

[3] Collected on January 6th, 2023

Table 5: The image size reduction per rule, note that the saving is computed at the image level where several smells could have been repaired.

Docker Smell	# Smell	Image Size Reduction			
		Total	Average	Median	Maximum
aptGetInstallUseNoRec	2242	188.1 GB (6.2%)	85.9 MB (6.2%)	17.1 MB (1.8%)	4.4 GB (87.7%)
pipUseNoCacheDir	2008	180.8 GB (7.1%)	92.2 MB (7.1%)	14.6 MB (1.6%)	6.7 GB (88.3%)
aptGetInstallThenRemoveAptLists	2170	163.2 GB (5.7%)	77 MB (5.7%)	13.4 MB (1.4%)	4.4 GB (87.7%)
npmCacheCleanAfterInstall	1640	46.5 GB (3.6%)	29 MB (3.6%)	1.3 KB (0%)	6.7 GB (88.3%)
tarSomethingRmTheSomething	184	4.6 GB (2.5%)	25.7 MB (2.5%)	318 Bytes (0%)	383 MB (38.7%)
yumInstallRmVarCacheYum	89	4.5 GB (4.8%)	51.2 MB (4.8%)	177 Bytes (0%)	801.4 MB (49.6%)
apkAddUseNoCache	887	3.5 GB (1.5%)	4 MB (1.5%)	628 Bytes (0%)	369.5 MB (32.7%)
mkdirUsrSrcThenRemove	219	2.6 GB (1.1%)	12.2 MB (1.1%)	275 Bytes (0%)	205.4 MB (32.7%)
gemUpdateSystemRmRootGem	34	877.4 MB (2.8%)	25.8 MB (2.8%)	306 Bytes (0%)	279.1 MB (17.6%)
gemUpdateNoDocument	31	863.5 MB (3%)	27.9 MB (3%)	948 Bytes (0%)	279.1 MB (17.6%)
npmCacheCleanUseForce	3	56.4 MB (10.1%)	18.8 MB (10.1%)	25 Bytes (0%)	56.4 MB (18.6%)
rmRecursiveAfterMktempD	2	43 Bytes (0%)	21.5 Bytes (0%)	0 Byte (0%)	43 Bytes (0%)
Total		277.03 GB (4.66%)	48.06 MB	1.9 MB	6.66 GB

Table 6: Impact of the smells on the bandwidth of DockerHub.

Docker Smell	# Docker Pull Per Week	Data saved per week
aptGetInstallUseNoRec	6 205 156	32.76 TB
pipUseNoCacheDir	3 591 566	8.62 TB
aptGetInstallThenRemoveAptLists	3 667 777	12.41 TB
npmCacheCleanAfterInstall	2 325 059	2.94 TB
tarSomethingRmTheSomething	440 604	42.07 GB
yumInstallRmVarCacheYum	675	10.45 GB
apkAddUseNoCache	2 048 579	664.63 GB
mkdirUsrSrcThenRemove	229 698	1.03 TB
gemUpdateSystemRmRootGem	8675	2.64 GB
gemUpdateNoDocument	266	2.64 GB
rmRecursiveAfterMktempD	319 649	4.1 MB
Total	11 784 785	40.45 TB

total and average weekly downloads on Dockerhub. Additionally, it shows the original image size, pull request ID, pull request status, number of repaired smells, image size reduction, and the theoretical average bandwidth saving per week (considering a median compression rate of 3.2 [36]).

Out of the 34 pull requests, 26 (76.5 %), were accepted and merged successfully including one required manual change. The remaining 6 (17.6 %) pull requests are awaiting responses from the developers. The accepted pull requests resulted in a total saving of 3.46 GB, which translates to a weekly saving of 2.05 TB, considering the 45 617 average weekly downloads.

Some pull requests triggered some discussions; while other pull requests were simply merged by developers without interaction. But most developers simply appreciated the contribution and merged the pull requests, as seen in PR-7 and PR-9. While those feedbacks do not explicitly address the Docker smells, it does indicate that developers value such contributions, suggesting they consider Docker smells as relevant. In a different case, developers explicitly expressed their interest in the changes, as seen in PR-11: *Hi, thank you very much, this change seems quite sensible! Cheers.*

Some other pull requests triggered additional discussions as illustrated in Figure 2. The developers asked if you could apply the changes to the base image of their application as seen in PR-18, some other repositories wanted to know about the tool that we use to create the fix such as PR-2.

During the discussions, there were instances of developers expressing concerns about specific repairs, notably regarding the *aptGetInstallUseNoRec* smell, which pertains to not installing recommended packages. For instance, developers in PR-17 were worried about the impact and asked us to remove that part of the change. Nonetheless, they appreciated the contribution and expressed gratitude for learning something new about Docker and `apt`.

Regarding the pull requests that were rejected, developers informed us that the changes were already present in the production or Alpine version as illustrated in Figure 3. In PR-27, the maintainer said *There's no need for this, just use the Alpine version.*, and in PR-28, they said *Duplicated. Already in production.*. While our pull requests were not merged, the fact that the changes were already implemented indicates that the smells are still considered relevant.

An important outcome of this research question is the acceptance by the developers. We have not faced yet a case where the developers do not find the change relevant. This is an interesting result compared to how smells are generally considered by developers. We expect that the perspective of the developers is different in this case because the impact of the smell can be directly measured, the number of false positives is low and the fix is comprehensive. This observation motivates us to extend further the ability of `Parfum` and to propose automatic patches for the Docker Smells.

Table 7: List of the pull requests that receive an answer from the maintainers. The complete list of opened pull requests is available in our repository [8].

#	Project	# Stars	# Image Pull		Image Size	PR ID	Status	# Smell	Data Saved	
			Total	Per Week					Image	per Week
1	AdWerx/pronto-ruby	20	198 057	1125	857.35 MB	171	Merged	6	38.68 MB (4.51%)	13.28 GB
2	pelias/openaddresses	46	90 188	304	577.31 MB	514	Merged	2	131.65 MB (22.8%)	12.2 GB
3	TomWright/mermaid-server	248	2172	15	889.97 MB	122	Merged	2	28.56 MB (3.21%)	136.32 MB
4	sqlfluff/sqlfluff	6876	43 313	743	208.15 MB	4262	Merged	3	11.74 MB (5.64%)	2.66 GB
5	rchakode/realopinsight	60	26 023	118	809 MB	30	Merged	2	39 MB (4.82%)	1.4 GB
6	vyperlang/vyper	4695	72 697	443	453.77 MB	3224	Merged	1	23.9 MB (5.27%)	3.23 GB
7	Kruptein/PlanarAlly	361	165 010	848	342.55 MB	1142	Merged	3	31.07 MB (9.07%)	8.04 GB
8	ShaneIsrael/fireshare	522	10 968	340	879.21 MB	166	Merged	4	158.71 MB (18.05%)	16.45 GB
9	jcraigk/kudochest	18	2096	28	1.52 GB	187	Merged	3	302.91 MB (19.42%)	2.57 GB
10	fzls/djc_helper	331	6682	95	489.2 MB	149	Merged	4	266.1 MB (54.39%)	7.68 GB
11	gotzl/accservermanager	48	7040	35	1.14 GB	53	Merged	1	680.38 MB (58.07%)	7.32 GB
12	nitrictech/cli	22	1042	34	1.47 GB	438	Merged	4	113.78 MB (7.55%)	1.19 GB
13	artsy/hokusai	89	396 984	1442	539.42 MB	323	Merged	2	10.07 MB (1.87%)	4.43 GB
14	brndnmtthws/tweet-delete	92	14 727	74	478.49 MB	107	Merged	2	19.94 MB (4.17%)	460.73 MB
15	bitovi/bitops	34	8496	70	168.45 MB	390	Merged	2	12.31 MB (7.31%)	270.08 MB
16	evennia/evennia	1671	37 540	121	1.25 GB	3091	Merged	5	195.49 MB (15.24%)	7.22 GB
17	sbs20/scanservjs	583	253 233	1846	1.04 GB	527	Merged	6	419.13 MB (39.45%)	236.15 GB
18	mitre/saf	118	4113	75	603.93 MB	989	Merged	2	124.01 MB (20.53%)	2.83 GB
19	w9jds/firebase-action	883	3 167 517	28 147	1.36 GB	176	Merged	4	124.99 MB (8.96%)	1.05 TB
20	naorlivne/terraformize	151	9663	57	131.89 MB	367	Merged	1	3.98 MB (3.02%)	70.82 MB
21	nwithan8/tauticord	78	1568	8	971.41 MB	60	Merged	1	18.89 MB (1.94%)	50 MB
22	azlux/botamusique	290	174 582	1316	667 MB	353	Merged	2	85.4 MB (12.8%)	34.31 GB
23	labsyspharm/scimap	46	3456	44	2.15 GB	43	Merged	1	307.74 MB (13.96%)	4.11 GB
24	leighmacdonald/gbans	32	1437	14	40.45 MB	374	Merged	3	2.46 MB (6.09%)	10.94 MB
25	alephdata/aleph	1881	2 254 039	8249	990.23 MB	2801	Merged	4	263.57 MB (26.62%)	663.54 GB
26	openedx/credentials	20	1893	26	1.28 GB	1912	Merged	8	132.96 MB (10.15%)	1.05 GB
27	atmoz/sftp	1469	982 418 661	2 289 101	155.55 MB	357	Closed	1	26.3 MB (16.91%)	17.94 TB
28	codacy/codacy-eslint	13	636 580	1685	1.38 GB	3741	Closed	2	195.24 MB (13.82%)	100.37 GB
34 Opened, 26 (76.5 %) Merged, 2 (5.9 %) Closed, 6 (17.6 %) Pending Pull Requests								78	3.46 GB	2.05 TB

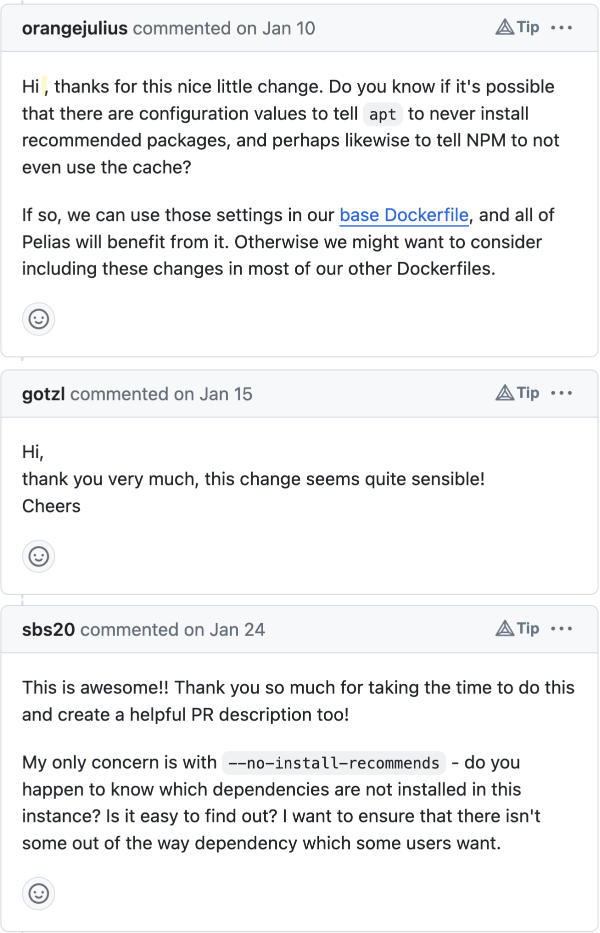
Answer to RQ3. We submitted 34 pull requests, 26 have been accepted, two have been rejected. Overall, the merged pull requests and the feedback that we received from the developers are overwhelmingly positive where developers acknowledged the fix even in the case of rejected pull requests. This observation contrasts with the general treatment that code smells are receiving from developers which highlights the importance and relevance of this study.

5 RELATED WORK

Docker has become a popular tool for developers and organizations to package, deploy, and run applications in a lightweight, portable container. As such, there has been a significant amount of research focused on improving the efficiency, security, and maintainability of Docker-based projects. In this section, we review several relevant studies that are related to this contribution.

A large number of papers studied the Docker ecosystem. We present a selection of them. Ibrahim et al. [17] investigate the number and diversity of images available on DockerHub for the same system, finding that there is a large number of images to choose from and significant differences between them. Ksontini et al. [18] study the occurrence of refactorings and technical debt in Docker projects, finding that refactorings are common but technical debt is rare. Xu et al. [33] present a study of mining container image repositories for software configuration information, finding that such information is often incomplete or outdated. Lin et al. [22] study the Docker images hosted on DockerHub. They observe a downward trend of Docker image sizes and smells in Dockerfiles. However, they also observed an upward trend in using obsolete base images. Lui et al. [23] also study DockerHub but focused on the security risks associated with it. Eng et al. [10] did a longitudinal study of the evolution of Dockerfiles, and they confirm that there are slightly fewer smells over time. However, none of those papers study the impact of the smells on the Docker image size.

Other works also focus on improving the security of containers, such as SPEAKER [20], which reduces the number of available system calls to a given application container by customizing and differentiating its necessary system calls at the booting and the



orangejulius commented on Jan 10

Hi , thanks for this nice little change. Do you know if it's possible that there are configuration values to tell `apt` to never install recommended packages, and perhaps likewise to tell NPM to not even use the cache?

If so, we can use those settings in our base Dockerfile, and all of Pelias will benefit from it. Otherwise we might want to consider including these changes in most of our other Dockerfiles.

gotzl commented on Jan 15

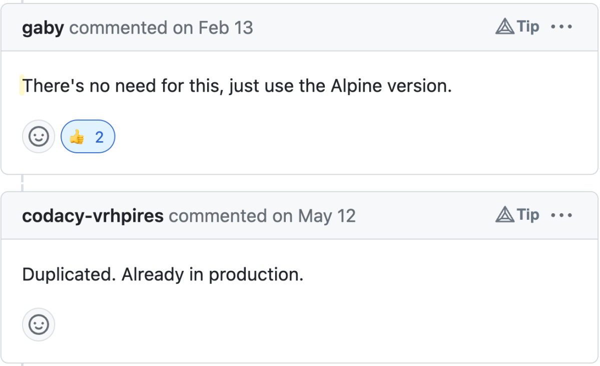
Hi,
thank you very much, this change seems quite sensible!
Cheers

sbs20 commented on Jan 24

This is awesome!! Thank you so much for taking the time to do this and create a helpful PR description too!

My only concern is with `--no-install-recommends` - do you happen to know which dependencies are not installed in this instance? Is it easy to find out? I want to ensure that there isn't some out of the way dependency which some users want.

Figure 2: Comment examples of merged PRs: PR-2,PR-11,PR-17



gaby commented on Feb 13

There's no need for this, just use the Alpine version.

codacy-vrhpires commented on May 12

Duplicated. Already in production.

Figure 3: Comments of rejected PRs: PR-27,PR-28

running phases. Confine [11] is a similar technique that uses static analysis to identify the required system calls.

Other contributions aim to improve or fix Dockerfiles. Henkel et al. [16] propose an approach for repairing Dockerfiles that do not build correctly. It uses machine learning to infer repair rules based on build log analysis. Hassan et al. [14] present Rudsea, a technique that adapts Dockerfiles based on the changes in the rest of the project. Zhang et al. [34] propose a technique that recommends Docker base images to improve efficiency and maintainability. Other tools aim to reduce the size of the Docker images by identifying bloat in the images and removing it. Cimplifier [25] and their framework [26] aim to automatically partition containers into simpler containers based on user-defined constraints. The goals are isolation of each sub-container, communicating as necessary, and only including enough resources to perform their functionality. strip-docker-image [29], minicon [12] and docker-slim [28] are open-source projects that reduce Docker image size by specializing the container to the application.

An important part of the bloat comes from bad practices. Several tools and works focus on identifying those Docker smells. Binnacle [15] is a tool for detecting Docker smells, they compared the presence of those smells between a set of Dockerfiles from GitHub and a set of Dockerfiles written by experts. They observed that there are five times fewer smells in the export Dockerfiles. Wu et al. [31] study the docker smell occurrence in 6334 projects. They show that smells are very common and there exists co-occurrence between different smells. Xu et al. [32] propose a technique based on static and dynamic analysis to detect temporary files inside Dockerfiles. Nonacademic works focus on detecting Dockerfile smells: Hadolint [13], dockerfilelint [6], docker-bench-security [5], or dockle [7]. However, none of these tools aim to repair the detected smells or analyze the impact of those smells.

Overall, there has been a significant amount of research focused on Docker, including tools for debloating, optimizing, and securing containers, as well as studies of the evolution and management of Dockerfiles and images. However, this empirical study is as far as we know the first that studies the impact of the smells on the Docker images and that collects feedback from the developers.

6 THREATS TO VALIDITY

In this section, we explore potential threats to the validity of our study and detail the measures taken to address them, thereby bolstering confidence in our results. Our classification framework aligns with the model proposed by Wohlin et al. [30].

6.1 Construct Validity

Construct validity threats stem from the alignment between theory and observation, largely influenced by the measurement procedures in our study. To address this, we took a meticulous approach. Firstly, we selected smell types reported and measured by a different research group. These smells were presented to practitioners, and their impact was measured. Another potential threat arises from the study's limited scope, focusing on specific smells in bash Dockerfiles. We mitigated this by verifying the presence of smells in recent Dockerfiles and presenting them to developers through pull requests.

Additionally, our focus on bash Dockerfiles excludes those in PowerShell, but given the prevalence of bash in Dockerfiles, our results remain relevant for the majority of users. We aimed to eliminate potential bias or subjectivity in the technique selection process.

6.2 Internal Validity

Internal validity focuses on establishing a reliable causal relationship between a treatment or intervention and its observed outcomes. One potential threat is the presence of internal bugs in Parfum. To address this, extensive testing was conducted, and Parfum was made open-source, enabling scrutiny by developers and researchers. Another potential threat involves the diversity of our dataset. To mitigate this, we collected a large and diverse dataset of Dockerfiles and supplemented our pull request selection with an existing dataset (Binnacle).

6.3 External Validity

External validity concerns the generalizability of study results. To enhance external validity, experiments were conducted on diverse case studies from different open-source projects, spanning various languages and sizes. While we focused on measuring the impacts of smells in terms of size and bandwidth on Docker images, this might limit the generalization of our results to all smell types. However, this specific impact aligns with common effects of smells, as supported by [15]. The relevance of image size in distributed systems further strengthens the importance of considering size increases in the evaluation of distributed software.

6.4 Conclusion Validity

Threats to conclusion validity involve the connection between the treatment and outcome, specifically regarding the reproducibility of the study's findings. To address this concern, we conducted experiments with a rigorous and mostly automated methodology. We also evaluated the precision and recall of the tool used in the study to ensure that our observations are reproducible. This comprehensive approach provides ample evidence to draw valid conclusions. Moreover, to ensure replicability, a rigorous methodology was followed in performing the experiments. The source code, scripts, and procedures are thoroughly documented, enabling other researchers to replicate the study with precision.

7 CONCLUSION

In this paper, we present an empirical study of the impact of Docker smells on image size. For this study, we identify and repair 16 145 Docker smells from 21 165 Dockerfiles.

We observe that smells lead to an average increase in image size by 4.66 % and a total of 40.45 TB of transfer per week (on DockerHub). Interestingly, the most common smells are related to the package managers and they are the smells that impact the most the image size.

Additionally, we verify the relevance of the smells by opening 34 pull requests on open-source projects that fix the Docker smells and reduce the Docker image size. We found that the developers react overwhelmingly positively to the pull requests by merging 26 (76.5 %) and by providing feedback that confirms that the smells are relevant to them even in the two cases where our pull requests were rejected.

The detection and repair of the smells has been performed by our tool Parfum. This study consequently also highlights the relevance of such a tool to help practitioners improve their Dockerfiles and therefore their Docker image.

Those results motivate us to continue this line of research and improve the Docker ecosystem. In particular, we aim to extend Parfum to support additional smell, integrate it inside IDE for direct developer feedback and also work on estimating the impact of the smells without having to build the Docker images.

DATA AVAILABILITY

We provide the scripts, dataset, and tool used in this contribution. You can find Parfum at [9], and the empirical study data at [8] as well as a functional demo of Parfum at https://durieux.me/docker-parfum.

REFERENCES

[1] Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for x86 virtualization. *ACM Sigplan Notices* 41, 11 (2006), 2–13.

[2] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 332–343. https://doi.org/10.1145/2970276.2970347

[3] 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, IEEE, New York, NY, USA, 323–333.

[4] William G Cochran. 1977. *Sampling techniques*. John Wiley & Sons, USA.

[5] docker-bench security. 2022. docker-bench-security: script that checks Docker deployment best practices. https://github.com/docker/docker-bench-security.

[6] dockerfilelint. 2020. hadolint: An opinionated Dockerfile linter. https://github.com/replicatedhq/dockerfilelint.

[7] dockle. 2020. dockle: Container Image Linter for Security. https://github.com/goodwithtech/dockle.

[8] Thomas Durieux. 2023. Open-science repository for the experiments of Parfum. https://doi.org/10.5281/zenodo.10439580 GitHub Repo: https://github.com/tdurieux/docker-parfum-experiment.

[9] Thomas Durieux. 2023. Open-science repository for Parfum. https://doi.org/10.5281/zenodo.10439571 GitHub Repo: https://github.com/tdurieux/docker-parfum.

[10] 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, New York, NY, USA, 449–459.

[11] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. 2020. Confine: Automated System Call Policy Generation for Container Attack Surface Reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 443–458. https://www.usenix.org/conference/raid2020/presentation/ghavanmnia

[12] grycap. 2020. Minimization of the filesystem for containers. https://github.com/grycap/minicon.

[13] hadolint. 2022. hadolint: Dockerfile linter validate inline bash written in haskell. https://github.com/hadolint/hadolint.

[14] Foyzul Hassan, Rodney Rodriguez, and Xiaoyin Wang. 2018. RUDSEA: Recommending Updates of Dockerfiles via Software Environment Analysis. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 796–801. https://doi.org/10.1145/3238147.3240470

[15] Jordan Henkel, Christian Bird, Shuvendu K. Lahiri, and Thomas Reps. 2020. Learning from, Understanding, and Supporting DevOps Artifacts for Docker. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 38–49. https://doi.org/10.1145/3377811.3380406

[16] Jordan Henkel, Denini Silva, Leopoldo Teixeira, Marcelo d'Amorim, and Thomas Reps. 2021. Shipwright: A Human-in-the-Loop System for Dockerfile Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, New York, NY, USA, 1148–1160.

[17] Md Hasan Ibrahim, Mohammed Sayagh, and Ahmed E. Hassan. 2020. Too many images on DockerHub! How different are images for the same system? *Empir. Softw. Eng.* 25, 5 (2020), 4250–4281.

[18] 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, New York, NY, USA, 781–791.

[19] Krishan Kumar and Manish Kurhekar. 2016. Economically efficient virtualization over cloud using docker containers. In *2016 IEEE international conference on cloud computing in emerging markets (CCEM)*. IEEE, IEEE, New York, NY, USA, 95–100.

[20] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. 2017. SPEAKER: Split-phase execution of application containers. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, New York, NY, USA, 230–251.

[21] 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, IEEE, New York, NY, USA, 371–381. https://doi.org/10.1109/ICSME46990.2020.00043

[22] 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, New York, NY, USA, 371–381.

[23] Peiyu Liu, Shouling Ji, Lirong Fu, Kangjie Lu, Xuhong Zhang, Wei-Han Lee, Tao Lu, Wenzhi Chen, and Raheem Beyah. 2020. Understanding the Security Risks of Docker Hub. In *Computer Security – ESORICS 2020*, Liqun Chen, Ninghui Li, Kaitai Liang, and Steve Schneider (Eds.). Springer International Publishing, Cham, 257–276.

[24] Ruben Opdebeeck, Jonas Lesy, Ahmed Zerouali, and Coen De Roover. 2023. The Docker Hub Image Inheritance Network: Construction and Empirical Insights. In *23rd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2023)*. IEEE, IEEE, New York, NY, USA, 198–208.

[25] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 476–486.

[26] Vaibhav Rastogi, Chaitra Niddodi, Sibin Mohan, and Somesh Jha. 2017. New Directions for Container Debloating. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation* (Dallas, Texas, USA) *(FEAST '17)*. Association for Computing Machinery, New York, NY, USA, 51–56. https://doi.org/10.1145/3141235.3141241

[27] 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* 32, 6, Article 142 (sep 2023), 30 pages. https://doi.org/10.1145/3603111

[28] SlimToolkit. 2023. Inspect, Optimize and Debug Your Containers. https://github.com/slimtoolkit/slim.

[29] Mark van Holsteijn. 2018. Utility to strip Docker images to their bare minimum size. https://github.com/mvanholsteijn/strip-docker-image.

[30] Claes Wohlin, Per Runeson, Martin H"ost, Magnus C Ohlsson, Bj"orn Regnell, and Anders Wessl'en. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[31] 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.

[32] Jiwei Xu, Yuewen Wu, Zhigang Lu, and Tao Wang. 2019. Dockerfile tf smell detection based on dynamic and static analysis methods. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, New York, NY, USA, 185–190.

[33] Tianyin Xu and Darko Marinov. 2018. Mining container image repositories for software configuration and beyond. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. Association for Computing Machinery, New York, NY, USA, 49–52.

[34] Yinyuan Zhang, Yang Zhang, Xinjun Mao, Yiwen Wu, Bo Lin, and Shangwen Wang. 2022. Recommending Base Image for Docker Containers based on Deep Configuration Comprehension. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, New York, NY, USA, 449–453. https://doi.org/10.1109/SANER53432.2022.00060

[35] Nannan Zhao, Hadeel Albahar, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Lukas Rupprecht, Ali Anwar, and Ali R. Butt. 2020. DupHunter: Flexible High-Performance Deduplication for Docker Registries. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, USA, 769–783. https://www.usenix.org/conference/atc20/presentation/zhao

[36] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Arnab K Paul, Keren Chen, and Ali R Butt. 2020. Large-scale analysis of docker images and performance implications for container storage systems. *IEEE Transactions on Parallel and Distributed Systems* 32, 4 (2020), 918–930.

[37] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S Warke, Mohamed Mohamed, and Ali R Butt. 2019. Large-scale analysis of the docker hub dataset. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, IEEE, New York, NY, USA, 1–10.