Empirical Software Engineering (2025) 30:150
https://doi.org/10.1007/s10664-025-10680-8

Mining and measuring the impact of change patterns for improving the size and build time of docker images

Giovanni Rosa[1] · **Emanuela Guglielmi**[1] · **Mattia Iannone**[1] · **Simone Scalabrino**[1] · **Rocco Oliveto**[1]

Accepted: 27 May 2025

Abstract
Software containerization, for which Docker is the reference tool, is widely adopted in modern software engineering. The performance of the Docker build process in terms of image size and build time is crucial to developers. While previous work and Docker itself provide best practices to keep the images small and fast to build, we conjecture that developers might adopt undocumented practices. In this paper, we present an empirical study in which we aim (i) to mine the practices adopted by developers for improving the image size and build time, and (ii) to measure the impact of such practices. As for the mining study, we manually analyzed a total of 1,026 commits from open-source projects in which developers declared they wanted to improve the image size or build time. We categorize such changes and define a taxonomy of 46 optimization strategies, including practices such as removing temporary files (*e.g.,* package manager cache) or improving the structure of the Dockerfile (*e.g.,* using multi-stage build). Such a taxonomy reveals some previously undocumented techniques, providing valuable insights for developers. As for the measurement study, we empirically assess the actual improvement in image size and build time (over 20 builds) of the most frequent change patterns observed in the mining study. Our results show that `changing the base image` has the best results in terms of image size, but it negatively affects the build time. On the other hand, we observed no change pattern that significantly reduces the build time. Our study provides interesting insights for both tool makers who want to support practitioners in improving Dockerfile build performance and practitioners themselves, who can better decide how to optimize their Dockerfiles.

Communicated by: Philipp Leitner

✉ Simone Scalabrino
simone.scalabrino@unimol.it

Giovanni Rosa
giovanni.rosa@unimol.it

Emanuela Guglielmi
emanuela.guglielmi@unimol.it

Mattia Iannone
m.iannone2@studenti.unimol.it

Rocco Oliveto
rocco.oliveto@unimol.it

[1] University of Molise, Campobasso, Italy

Published online: 01 August 2025

Keywords Dockerfile smells · Empirical software engineering · Software evolution

1 Introduction

Containerization technologies are widely adopted in the modern era of software engineering, allowing to speed up the deployment and release process for application context (Bernstein 2014). Docker is the leading platform for containerizing software application, resulting the most used and desidered tool in the recent StackOverflow survey.[1][2] Docker allows to easily wrap applications along with the required dependencies for their execution, ensuring to share them by different systems and execution environments.

Having small Docker images, in terms of storage size, is desirable because it allows using less resources on the deployment server and, thus, reduce the deployment costs. At the same time, reducing the time needed to build an image from the source Dockerfile is important in a scenario in which developers frequently deploy their product (*e.g.,* in a DevOps environment). Open-source tools are available for improving the performance of Docker artifacts, specifically to reduce the size of containers.[3] However, such tools work directly on the Docker images and they need to be executed every time a new build is completed. Ideally, the source Dockerfile should be reasonably optimized already to avoid such an overhead.

The literature confirms that both such performance-related aspects are important to developers. Rosa et al. (2023) showed that developers prefer smaller images. Besides, Zhang et al. (2018) report that slow build time in CI/CD pipelines (which includes the build of Dockerfiles) lead to poorer developers' work efficiency. Ksontini et al. (2021) found that ∼19.7% of the Dockerfile refactoring operations performed by developers are aimed at improving such aspects. While such a study provides valuable insights on the practices used by developers to improve both the build time and the size of Docker images, its goal was more generic (*i.e.,* it was aimed at studying all the refactoring operations performed by developers). Thus, the authors ended up analyzing only 38 commits related to performance improvement. Besides, we do not know the impact of refactoring operations made by developers. We conjecture that such a previous work only scratched the surface of what developers do to address performance issues in Docker images.

In this paper, we present an extensive empirical study in which we aim at understanding (i) what developers do to improve the image size and build time of Docker images, and (ii) what impact such operations have in practice. Starting from the state-of-the-art dataset proposed by Eng and Hindle (2021), containing commits aimed at modifying Dockerfiles in GitHub, we run two queries to extract the changes that are aimed at improving either the build time or the image size. Such queries were defined by selecting relevant keywords from the dictionary of unique words in the commit messages we considered. We manually analyzed a significant sample of 905 commits, with the aim of manually tagging them with a high-level description of the operations made by developers. As a result, we selected a total of 905 commits (1,230 tags). Next, we defined a taxonomy of 46 refactoring operations made by Dockerfile developers to reduce the image size and build time of the resulting Docker images. We found that most of the changes are aimed at de-bloating the image (49% of the commits analyzed) by removing unnecessary files and, thus, reducing the image size. Besides,

1 https://survey.stackoverflow.co/2023/

2 https://survey.stackoverflow.co/2024/

3 https://github.com/slimtoolkit/slim

developers tend to modify the Dockerfile architecture (36% of the changes) and to foster the use of caching mechanisms (18% of the changes) to reduce the build time. As a second contribution, we conducted an experiment to measure the impact of the most frequent change practices we found in the first part of the study on both image size and build time. To do this, we first selected the commits for which the build of the Dockerfile before the change succeeded. Then, for each selected commit, we considered the difference between the Dockerfile after the performance-improving change and manually extracted several alternative versions of the Dockerfile, each of which implemented exactly a performance improvement practice. For example, if a commit changed both the base image and joined RUN instructions, we extracted two improved versions: one in which we only changed the base image and one with the previous base image but with RUN instructions joined. Then, we built both the previous version of the Dockerfile and each new improved version to measure build time and image size. We repeated each build 20 times to account for the variability of build time. Finally, we analyzed the improvements (both in terms of build time and image size) obtained with each fixing pattern. We found that changing the base image variant has the most substantial impact in reducing the image size (62% reduction, on average). Changing the base image altogether, instead, has a more limited reduction on image size (15% reduction, on average). Other positive practices include enabling multi-stage builds and joining multiple `RUN` instructions, which effectively reduce the number of image layers and help shrink the image size, on average, by 62% and 24%, respectively. As for the build time, the results are more nuanced. While some practices, such as joining `RUN` instructions, showed positive effects in some cases, others did not lead to consistent improvements. No category allowed us to obtain statistically significant improvements in terms of build time. Instead, changing the base image variant results in significantly higher build time (+131%).

Our results might be useful to both developers and researchers. We provide developers with a catalog of changes they might apply to reduce the image size and build time of Docker images. Researchers might use our results to devise approaches to support developers in the task of refactoring Docker images for performance improvement (*e.g.,* by automatically suggesting optimal base images).

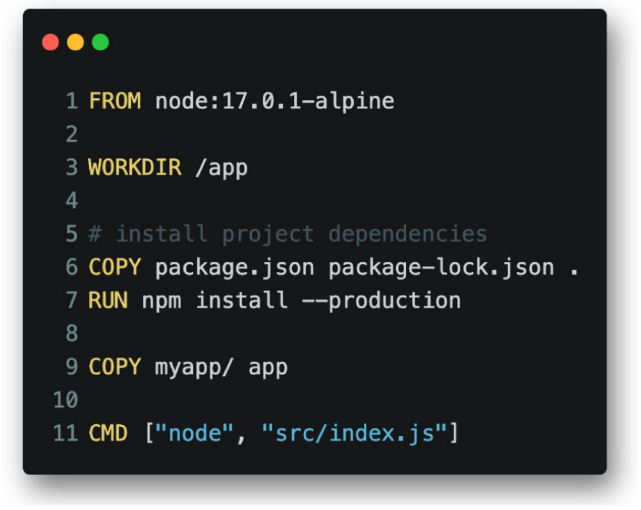
The rest of our paper is organized as follows. In Section 2, we comprehensively review the relevant background literature. Section 3 presents the methodology and details of our empirical study, including the data collection process, experimental design, and techniques applied. In Sections 4 and 5 we report and discuss the results, followed by threats to validity in Section 6. Finally, in Section 7 we conclude the paper and provide future directions.

2 Background and Related Work

Docker images are the result of the building process of Dockerfiles. Each Dockerfile starts with a *base image*, then follows a series of instructions defining requirements and configurations. The build process converts each Docker instruction into a binary layer in consecutive order. This means that images are composed of a series of stacked individual layers, each one dependent on the previous one. To optimize the re-build process of Dockerfiles, only the changed layers are rebuilt while those already built, if not changed, are reused reducing the total build time required. On the other hand, if a layer is changed, all the following layers will be invalidated and rebuilt due to the fact that each layer depends on the previous one. Cache invalidation occurs for all the instructions modifying the filesystem (e.g., `RUN`, `COPY`, `ADD`) or changing the execution environment (*e.g.,* `ENV`, `ARG`, `ENTRYPOINT`). For example, if a

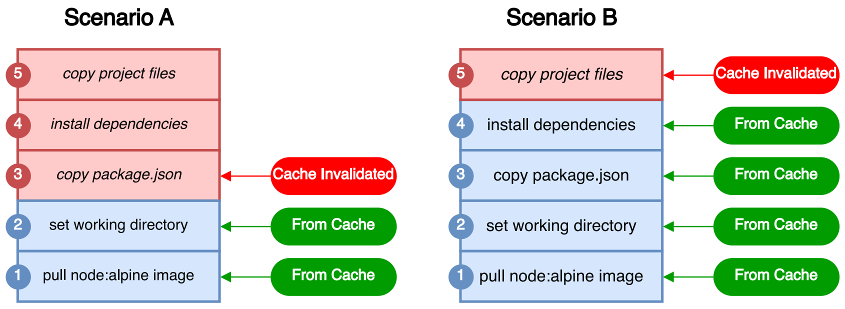
COPY instruction is changed, all the layers that depend on it will be invalidated and rebuilt. This is a common scenario when the source code of the application changes. Figure 1 reports a detail of the layer caching system. The example reports a Dockerfile using node.js. Thus, when the dependencies of `package.json` file are changed, the image will be rebuilt from that point (Scenario A). Instead, if only the source of the application is changed, only the cache of the final layer is invalidated and then rebuilt (Scenario B).

The number of layers can directly impact the final size of the Docker image. In some cases, a large size could be a symptom of bad quality (Rosa et al. 2023). The most widespread instrument used for identifying quality issues in Docker artifacts are Dockerfile smells (Wu et al. 2020). Dockerfile smells are violations of best writing practices in Dockerfiles that can negatively impact the resulting images in terms of size increase, security, and reliability issues (Cito et al. 2017). Several studies investigated the occurrences over time of smells (Lin et al. 2020; Eng and Hindle 2021). Even if smells are widely diffused, there is a declining trend in their occurrence and also in the size of images. This aspect is also reported as technical



```dockerfile
FROM node:17.0.1-alpine

WORKDIR /app

# install project dependencies
COPY package.json package-lock.json .
RUN npm install --production

COPY myapp/ app

CMD ["node", "src/index.js"]
```

(a) Dockerfile



(b) Workflow of Dockerfile build when the dependency files change (Scenario A) and the application files change (Scenario B).

Fig. 1 Example of Dockerfile using node.js (top), and an example of how the layer caching works (bottom)

debts by developers (Azuma et al. 2022), specifically in terms of files and dependencies that should be removed. This means that developers pay attention to the image size and wasteful resources in Dockerfiles.

Previous works proposed approaches to improve the quality of Dockerfiles (Henkel et al. 2021; Bui et al. 2023), and, specifically, to fix smells (Rosa et al. 2024; Durieux 2024). It has been proved that fixing smells can reduce the space wastage of containers (Durieux 2024). Another important aspect, still regarding performance and related to poor design choices, is the build time of Docker images. In fact, previous studies reported that developers keep attention to this aspect as they are led to change their CI/CD pipelines due to the slow build speed of the embedded Docker images (Zhang et al. 2018). However, none of these studies performed an extensive investigation on what are the changes that can help to improve the build time and image size of Docker images.

The only exception is the study conducted by Ksontini et al. (2021). They investigated a sample of refactoring operations performed in Dockerfiles and *docker-compose* files, which define multi-container applications and are commonly used to orchestrate multiple services during development and deployment. In particular, they manually investigated a total of 193 commits where only 28 of them regard image size, while 10 the build time. Therefore, our study aims to perform a more in-depth analysis of those two particular aspects.

Other approaches have been presented specifically for container debloating. Examples are the works of Skourtis et al. (2019) and Jiang (2022) that work directly on containers to reduce layer redundancy and space wastage. In addition, Rastogi et al. (2017a, b) proposed techniques leveraging dynamic analysis to identify only the necessary resources of a given container in order to allow removing the unnecessary ones. We believe that working at Dockerfile-level allows developers to have more control over the resulting Docker images, avoiding unwanted side effects when working directly on the final containers. It is worth saying that our study aims to quantitatively measure the impact of the changes applied to improve performance, and thus our results provide a set of recommendations of which changes developers should apply in order to make those improvements.

3 Empirical Study Design

The *goal* of our study is to understand how Docker developers change Dockerfiles to improve the build time and size of the resulting Docker images and how such changes impact those quality aspects. The *perspective* is of both researchers and developers interested in improving those aspects when writing Dockerfiles. The *context* consists of 905 commits coming from 977 open-source repositories.

In detail, the study addresses the following research questions:

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

We want to investigate the developers' activity on existing Dockerfiles to capture the common change patterns applied to improve the image size and build time of the resulting images.

RQ_2: To what extent do performance improvement changes impact image size and build time of Docker images?

With this RQ, we aim to measure the impact of individual changes on image size and build time. This analysis allows us to identify which changes produce improvements and which may introduce tradeoffs or negative effects.

3.1 Data Collection

The context of our study is represented by commits extracted from the dataset proposed by Eng and Hindle (2021). While other Dockerfile datasets exist (Lin et al. 2020; Henkel et al. 2020), this is, currently, the largest one in the literature. It contains the change history of Dockerfiles extracted from all the open-source GitHub repositories up to 2021. The data extracted regard a total of 1.9M repositories, for a total of 11.5M commits related to about 9.4M Dockerfiles. We chose this dataset because it provides both breadth (large and diverse project base) and depth (historical commit-level changes to Dockerfiles). Unlike other datasets (such as Henkel et al. (2020) which contains a subset of Dockerfiles specific for writing patterns, Lin et al. (2020) which contains only Dockerfiles related to Docker Hub repositories, and Zerouali et al. (2021) which contains only Debian-based images), the one provided by Eng and Hindle (2021) includes commit-level evolution traces (*i.e.,* both commits and modified files) specifically related to Dockerfiles, enabling us to reconstruct and analyze the changes performed by developers over time. We needed a dataset with detailed commit histories to (i) extract meaningful change patterns, (ii) ensure that the modifications were made by developers themselves with the intent of improving performance, and (iii) is representative of the development activities in open-source repositories. For our study, we selected a subset of those changes, composed of the commits aimed at reducing the build time and image size of the resulting Docker images. To achieve this, we rely on the commit message, *i.e.,* we select the commits in which developers explicitly report the intention of improving the build time or reducing the size of the Docker images. To extract performance-related changes, we applied a keyword-based filtering heuristic (described below), targeting commits whose messages explicitly reference improvements to image size or build time. We manually validated these commits to ensure they include meaningful Dockerfile changes, as discussed later in Section 3.2. While our sample is not exhaustive, it is designed to be representative of real-world, performance-oriented Dockerfile modifications.

We defined a heuristic approach to filter commits based on what developers reported in the commit message using Natural Language Processing (NLP) techniques. In particular, we defined a *Python* NLP pipeline using the *Spacy*[4] tool. The pipeline is composed of two phases: a text pre-processing phase, followed by a keyword-based query to select only the relevant commits. The entire process is reported in Fig. 2 and detailed below.

Text Pre-processing First, we split the plain text in commit messages into single tokens (*i.e.,* words). We achieve this by applying *word tokenization*. Follows a *stop-word* filtering, in which the non-informative tokens (such as articles and conjunctions) are removed. As a last step, we apply a lemmatization procedure that reduces each word to the root form (*e.g., changing* becomes *change*) maintaining the original meaning. This results in a list of tokens that can be analyzed in the next step.

[4] https://spacy.io/

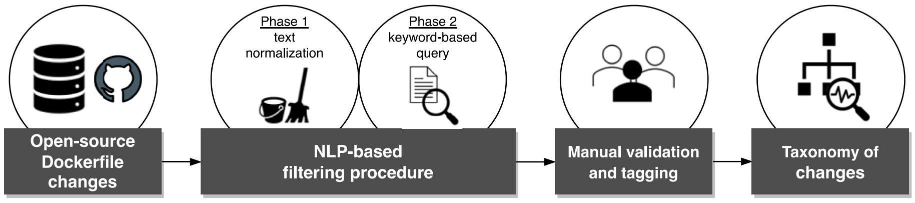

Open-source Dockerfile changes
Phase 1
text normalization
Phase 2
keyword-based query
NLP-based filtering procedure
Manual validation and tagging
Taxonomy of changes

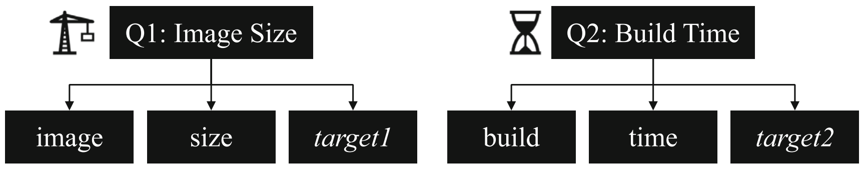
Fig. 2 A summary of the experimental procedure applied to extract the performance-related changes analyzed in our study

Keyword Selection To filter only the performance-related commits improving build time and reducing Docker image size, we applied a keyword-based filter to the processed commit messages using two different queries, one for each scope. The queries have been defined via a manual process in which one of the authors extracted the dictionary from all the commit messages (*i.e.,* the unique words appearing in them) and selected the ones judged as relevant to indicate an improvement in each of the two aspects we focused on. In the end, we defined the two queries (Q1 and Q2) reported in Fig. 3.

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

3.2 Experimental Procedure

To answer $\mathbf{RQ_1}$, we applied our filtering procedure to all the commits contained in the dataset by Eng and Hindle (2021). Thus, we obtained a subset of 11,000 commits matching at least one of the two queries. We excluded all the commits that are no longer available. Also, one of the authors manually validated the commit messages, along with the changes, selected by the two queries excluding those that are (i) false positives wrongly selected by our heuristic



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

Fig. 3 The queries used to filter performance commits. The first two keywords are connected with an AND operator, while the other ones (*target1* and *target2*, respectively) are connected with an OR operator

approach (*e.g.,* commit[5] since in the message the author explicitly report: “[...] the resulting image is slightly larger”), (ii) not available anymore, and (iii) duplicated (*i.e.,* from forked repositories). In addition, the author double-checked and annotated if the commit improves the build time, decreases the image size, or both: Indeed, some of the commits we selected for one of the aspects were also aimed at improving the other one. In this step, our aim was to simply check if the *intention* of the developers was to improve performance, *i.e.,* if commit message explicitly mentioned the willing to reduce the build time, the image size, or both of the aspects on the performed change. We did this because the simple automated filtering approach we adopted could introduce false positives. For example, we discarded a commit[6] from *jrgm/fxa*. Even though it contained the matching keywords (“feat: *reduce build time* for fxa-email-service.”), the message did not refer to improved Dockerfile build time, but rather to a refactoring of the program code. Incidentally, the Dockerfile was modified as well. The process is performed one commit at a time, applying both queries, until a sample of 1,026 valid commit candidates is reached. This is in line with similar studies from the literature (Ksontini et al. 2021). Next, two of the authors manually annotated the commits with one or more tags to describe the type of change. The two annotators had a “fair” agreement rate ($\kappa = 0.4$) (Landis and Koch 1977). Next, we performed a cross-validation phase which involved both the two original annotators and an additional annotator (one of the authors) to discuss and resolve the conflicts. Note that, during the manual validation, some commits have been flagged as *not valid* (*i.e.,* the change can not concretely reduce the build time or the image size[7]), and thus excluded from the final set of commits. In the end, we obtain a total of 905 valid commits. The agreement rate between the resolution annotator and the two annotators resulted as “moderate” ($\kappa = 0.55$) (Landis and Koch 1977). We provide the final set of selected commits in the replication package (Rosa et al. 2024) to support future studies.

Finally, we use a card-sorting inspired approach (Spencer 2009) to categorize the tags we identified and organize them in a taxonomy. More specifically, the two annotators started with a first round aimed at abstracting the tags. Then, followed two more rounds in which they grouped the similar changes. In the end, they discussed the obtained tags and ordered them in macro- and sub- categories to build a first draft of the taxonomy. A final round followed in which the two annotators double-checked each tag and the assigned category, by renaming and reordering them when needed. After this, the final version of the taxonomy has been obtained. Note that this step was not performed independently, but collaboratively by the annotators. The disagreements in terms of categories to merge and position in the taxonomy were directly discussed and resolved in this phase. For each change, we report the number of occurrences and which aspect it improves (*i.e.,* build time, size, or both). We report and discuss in detail the taxonomy reporting some representative examples for each category. The workflow adopted is summarized in Fig. 2.

To answer **RQ$_2$**, we first select the categories of fixing patterns of interest and, consequently, the commits that we will focus on. Then, we manually isolate the performance-improving changes in the Dockerfiles to measure the improvement of each category. Finally, we built such Dockerfiles and measured the build time and image size. We report in details below each of such step.

[5] https://github.com/hypothesis/bouncer/commit/0313392

[6] https://github.com/amitmbee/krapp/commit/7c80f82

[7] https://github.com/cropgeeks/docker/commit/bb96380

Table 1 Categories of changes analyzed to answer RQ$_2$

Category	#Instances
change base image variant	26
choose a different base image	13
use multi-stage build	13
join RUNs	10
remove unused bin. and dep.	7
remove inst. dep. and src.	4
remove apt-get lists	4
apt-get clean-autoclean	3
remove apk cache	2
apk –no-cache	2
Total	84

Category Selection We base the selection of the categories for which we measure the impact in terms of image size and build time on two criteria: (i) theoretical adequacy (given our experimental setup), and (ii) data availability. As for the former, we first exclude all the categories related to the Docker cache (*i.e., move sources copy at bottom*, *copy/install requirements beforehand*, and *compile for fewer versions/targets*) since they are incompatible with the procedure we adopted to measure build time. Such changes reduce the build time only after the first build has been completed (since they improve the use of caching mechanisms). However, as we will explain later, we disabled Docker cache to measure the build time multiple times in a reliable way without biases. Second, we discarded the *extract Dockerfile as base image* category because it would have been necessary to design an ad-hoc procedure to first build the extracted Dockerfile and then the target one. Note that each extracted Dockerfile could have been at a different path, based on the project at hand. We filter out all the commits belonging to such discarded categories. Given the remaining commits belonging to the selected categories, we performed a test build of all the Dockerfiles *before* and *after* the change. We filtered out commits for which the build *before* the change failed (*i.e.,* 453 cases). When the build succeeded in the version *before* the change but failed in the version *after* the change (24 cases), we manually inspected and attempted to fix such Dockerfiles. We could do this for only two of them. The build errors in the other 22 Dockerfiles depended on outdated or missing resources (*e.g.,* unavailable base images or deprecated libraries). Thus, we discarded them. We also discarded a commit[8] because the new base image adopted resulted in an empty image which likely means that there this was part of a bigger change or a developers' mistake. Given the remaining commits after this filtering (*i.e.,* 92 instances), we re-counted the number of commits available for each category of change and further discarded the categories for which we ended up with less than two examples.[9] We did this because any conclusion deriving from a single measurement could be too project-specific and thus not sufficiently generalizable. We ended up with 10 categories, depicted in Table 1.

Isolating the Performance-Improving Changes Each performance-improving commit could tackle one or more categories. We want to measure the impact at the level of the *category* of change. Therefore, we needed to separate the changes belonging to different cat-

[8] https://github.com/razzkumar/todo/commit/0bfe035

[9] We further removed *remove unused packages*, *remove temporary files in /tmp/* /var/tmp/**, *remove-avoid dev dependencies*, *apt-get –no-install-recommends*, *remove layers*, *remove-avoid pip cache*, *build binaries and copy to container*, *apt-get purge-autoremove*

egories. For example, if a commit both changed the base image and joined `RUN` instructions, we wanted to have two improved version: one only with the new base image (with non-joined `RUN` instructions) and one with joined `RUN` instructions (but with the old base image). We manually defined alternative improved versions of the Dockerfiles (based on the *after* version actually written by the developers) for each commit to which we assigned more than one tag in $\mathbf{RQ_1}$. This allowed us to independently test the improvement of each category. As a result, for each commit, we have one original Dockerfile (D_0, *i.e.,* the one before the improvement) and n Dockerfiles (D_i, each one implementing a single performance-improving change).

Measuring Build Time and Image Size For each commit under analysis, we cloned the repository and checked it out at that specific snapshot. In this context, we iteratively replaced the Dockerfile under test with D_0 (the one without improvements) and each improved version D_i. For each of them, we ran a warm-up build of the Docker (to make the system download the base images) followed by 20 builds, for which we measured the build time and the resulting image size. We did this to account for the non-determinism of build time (the image size, on the other hand, is deterministic). In this step, we disabled the Docker cache to ensure consistent measurements as for the build time.

Statistical Analyses To study the impact on *image size*, we analyze, for each category C, the percentage of cases in which the size is reduced, increased, or did not change at all by comparing the size of D_0 with the size of the D_i of implementing the change C. In addition, we check the significance of the difference of each category by using the Wilcoxon Signed-Rank test (Woolson 2007). The null hypothesis is that the category of improvement has no effect on the image size. We reject the null hypothesis if the p-value is lower than 0.05. We also compute the effect size to quantify the magnitude of the significant differences we find. We use Cliff's Delta (Macbeth et al. 2011) since it is non-parametric. We use Cliff's delta lays in the interval [-1, 1]: The effect size is **negligible** for $|\delta| < 0.148$, **small** for $0.148 \leq |\delta| < 0.33$, **medium** for $0.33 \leq |\delta| < 0.474$, and **large** for $|\delta| \geq 0.474$.

To analyze the *build time* we used an analogous procedure. In this case, we have 20 measurements for each build. Given the build times of D_0 and a D_i, we check whether the change allows to significantly modify the build time by using the Wilcoxon Signed-Rank test (Woolson 2007). The null hypothesis here is that the specific modification did not impact the build time (the differences we observe are due to the chance). When the p-value is lower than 0.05, we say that single change *increases* or *reduces* the build time (based on the mean build time). All the other instances, instead, do not change the build time. We report, for each category, the percentage of cases in which the related changes improved, reduced, or did not affect the build time. Besides, similarly to what we do for image size, we consider the mean build time of each instance and use the Wilcoxon Signed-Rank test (Woolson 2007) to test if the category of changes significantly affects the build time. The null hypothesis is that the category of improvement does not affect such a variable. Again, we also report the Cliff's Delta for the categories that have a significant impact.

3.3 Technical Setup and Data Availability

To ensure consistency and reproducibility of the results reported in our analysis, all experiments (including Docker image builds and performance measurements) were executed on a cloud virtual machine exclusively allocated for this study. The machine was configured with the following specifications: Intel Broadwell (no TSX, IBRS) CPU with 8 cores at 2.2 GHz, 20 GB of RAM, and a 400 GB solid state drive (SSD). The operating system is Ubuntu 22.04.5 LTS, with kernel 5.15.0-25-generic. The software environment includes Bash 5.1.16, Python 3.11.11, Ruby 3.0.2p107, and Docker 28.0.1. All the scripts adopted to run the experiment are written in Python and Ruby. No other tasks or processes were executed concurrently on the machine during the experiment sessions to avoid any interference and ensure the stability and accuracy of time measurements. The execution of the whole measurement experiment took about one week on such a dedicated machine. We provided the tagged commit dataset, the NLP filter heuristic and the scripts used to run the measurement experiment in our replication package (Rosa et al. 2024).

4 Empirical Study Results

This section reports the analysis of the results for the two research questions of our study.

4.1 RQ_1: Mining Performance Changes

We report in Fig. 4 the taxonomy of changes applied by developers to reduce the build time and image size of Dockerfiles and Docker images. We report, for each category, the number of occurrences of that type of change. Moreover, we report the single change as an *attribute* having the number of specific occurrences and a badge indicating if the change reduces the image size (**S**), the build time (**T**), or both (**S** and **T**). We assigned a total of 1,230 tags grouped in 4 different macro-categories, as described in the following.

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

The most frequent changes are categorized as Debloating (49%), followed by Dockerfile Architecture (36%). Categories Caching and Tweaks are the less frequent (12%) and 4%). The changes in Debloating mainly impact the final image size, while those in Tweaks and Caching the build time. On the other hand, changes in Dockerfile Architecture impact both aspects. In the following we describe them in detail by reporting some examples.

4.1.1 Debloating

To reduce the clutter in Docker images, developers remove the additional data used by package managers (209 occurrences), *i.e.,* by performing a cleanup of the packages, data, and cache during the installation of dependencies. This kind of change mainly aims to reduce the image size. For example, when installing a dependency using the `apt` package manager, a common pattern is to run `apt-get clean`, remove `apt` lists and the used cache. In some cases, running additionally the command `apt-get autoremove` could also contribute to remove unnecessary files. We report in Fig. 5 an example for *remove* `apt-get` *lists* and `apt-get` *clean/autoclean* changes, proposed in commit.[10] Specifically, `apt-get clean`

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

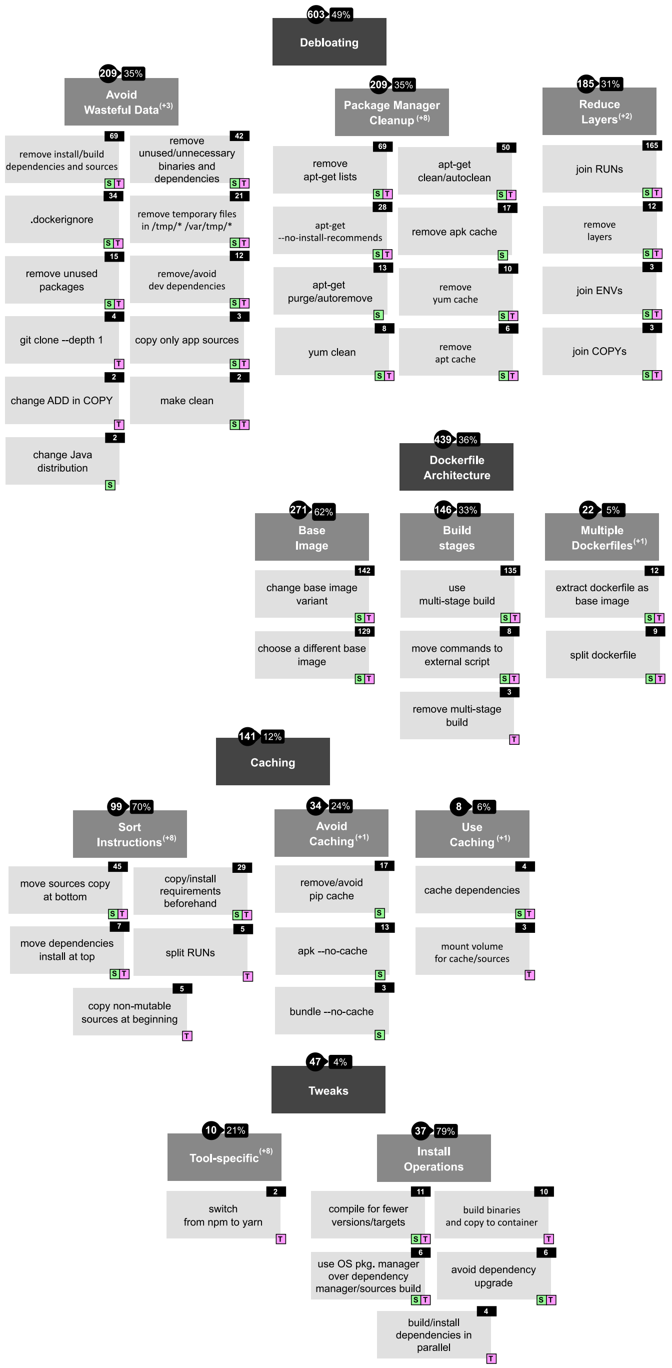
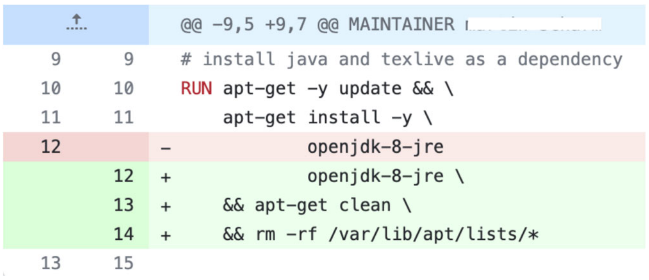

603 49% Debloating
209 35% Avoid Wasteful Data (+3)
69 remove install/build dependencies and sources S T
42 remove unused/unnecessary binaries and dependencies S T
34 .dockerignore S T
21 remove temporary files in /tmp/* /var/tmp/* S T
15 remove unused packages S T
12 remove/avoid dev dependencies S T
4 git clone --depth 1 T
3 copy only app sources S T
2 change ADD in COPY T
2 make clean S T
2 change Java distribution S
209 35% Package Manager Cleanup (+8)
69 remove apt-get lists S T
50 apt-get clean/autoclean S T
28 apt-get --no-install-recommends S T
17 remove apk cache S
13 apt-get purge/autoremove S
10 remove yum cache S T
8 yum clean S T
6 remove apt cache S T
185 31% Reduce Layers (+2)
165 join RUNs S T
12 remove layers S T
3 join ENVs S T
3 join COPYs S T
439 36% Dockerfile Architecture
271 62% Base Image
142 change base image variant S T
129 choose a different base image S T
146 33% Build stages
135 use multi-stage build S T
8 move commands to external script S T
3 remove multi-stage build T
22 5% Multiple Dockerfiles (+1)
12 extract dockerfile as base image S T
9 split dockerfile S T
141 12% Caching
99 70% Sort Instructions (+8)
45 move sources copy at bottom S T
29 copy/install requirements beforehand S T
7 move dependencies install at top S T
5 split RUNs T
5 copy non-mutable sources at beginning T
34 24% Avoid Caching (+1)
17 remove/avoid pip cache S
13 apk --no-cache S
3 bundle --no-cache S
8 6% Use Caching (+1)
4 cache dependencies S T
3 mount volume for cache/sources T
47 4% Tweaks
10 21% Tool-specific (+8)
2 switch from npm to yarn T
37 79% Install Operations
11 compile for fewer versions/targets S T
10 build binaries and copy to container T
6 use OS pkg. manager over dependency manager/sources build S T
6 avoid dependency upgrade S T
4 build/install dependencies in parallel T

Fig. 4 Taxonomy of changes reducing build time and image size for Dockerfiles and Docker images. The total number of occurrences are reported for each category and sub-category. Additionally, attributes have a badge indicating if the change reduces the image size (**S**), the build time (**B**), or both (**S** and **T**)



```
@@ -9,5 +9,7 @@ MAINTAINER [redacted]
  # install java and texlive as a dependency
  RUN apt-get -y update && \
      apt-get install -y \
-             openjdk-8-jre
+             openjdk-8-jre \
+     && apt-get clean \
+     && rm -rf /var/lib/apt/lists/*

```

Fig. 5 Example of a "Debloating" change, aimed at removing the apt cache and sources lists

and the removal of the sources lists have been added right after calling `apt-get install` to install packages.

Other types of changes are focused on the removal of wasteful data (209 occurrences). Examples are excluding development dependencies from the final Docker image or removing temporary files (*e.g.,* removing `/tmp/*` and `/var/tmp/*`). Another typical change is the addition of a `.dockerignore` file. Such a file contains patterns of files that should be excluded from the build context. This change has a positive impact on both the build speed (*i.e.,* smaller context to handle) and the size, as it might reduce the number of files that are copied in the image through `COPY` or `ADD` instructions.

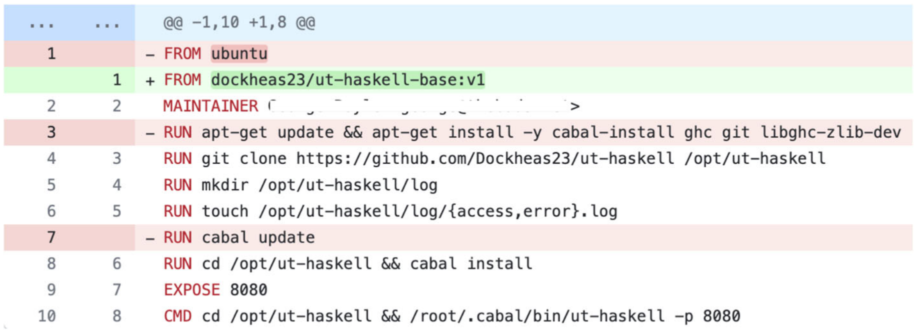
Finally, developers often aim at reducing the number of layers in the Docker image (185 occurrences) to reduce both the image size and the build time. To do this, in most of the cases, developers join several `RUN` instructions in a single one.

4.1.2 Dockerfile Architecture

The most frequent type of modification from this category is the change of the base image (271 occurrences). Developers usually prefer a variant of the same Docker image (142 occurrences), *e.g.,* `python:3.11-alpine` instead of `python:3.11`. A common example is the adoption of the *alpine* flavor of the same base image, which is typically smaller. Alternatively, they switch to a completely different base image (129 occurrences) because either it is smaller (*e.g.,* from `ubuntu` to `debian`) or reduces the build time because it already has the binaries for some required dependencies (*e.g.,* from the generic `ubuntu` to one already including `python`). Thus, the installation steps for those dependencies are removed from the Dockerfile reducing the overall build time. An example of a *Base Image* change is reported in Fig. 6 (from commit[11]). In detail, the generic `ubuntu` base image is replaced with a more specific one containing already the required `haskell` dependencies, avoiding installing them after in the Dockerfile.

Another frequent operation is adding or modifying *Build Stages* (146 occurrences) of the Dockerfile. In this case, developers more often restructure the Dockerfile enabling multi-stage builds (135 occurrences). This consists of grouping the Dockerfile instructions in separate stages, corresponding to isolated steps of the build process executed in a new Docker image. This allows to discard temporary files, reducing the size of the final image (used as the final step), and easily handling the build dependencies (*e.g.,* using a pre-built image) reducing also the build time.

[11] https://github.com/vmware-archive/kubeless-ui/commit/6741125



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

Fig. 6 Example of a "Dockerfile Architecture" change in which the base image is replaced with one including `haskell` dependencies

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

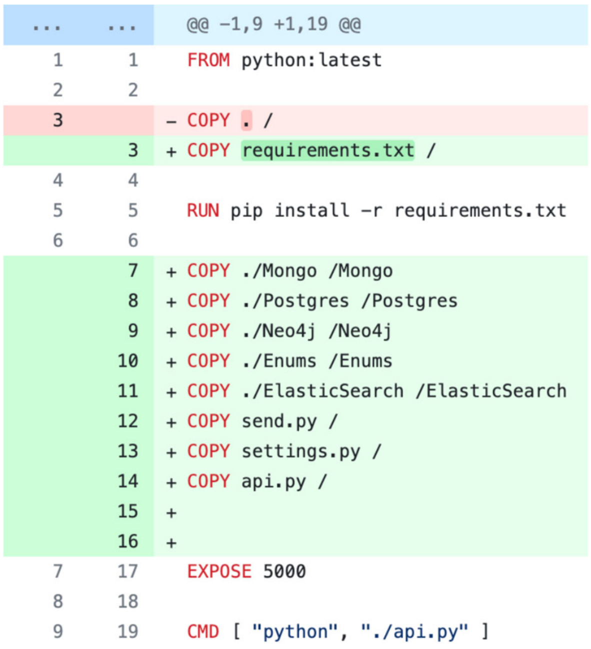
4.1.3 Caching

The changes in this category consist mainly of modifying the instruction order (Sort Instructions, 99 occurrences) to improve the usage of the layer caching during the build. This means, for example, moving the COPY instruction to the bottom of the Dockerfile (45 occurrences). Since the source files are those that usually change, it will result in a faster build, especially during development. Another common operation is to copy and install the requirements before copying sources or performing other operations (29 occurrences), in order to reduce the build time. A common example (Fig. 7) is to copy only the Python `requirements.txt` before installing the requirements separately from the other sources. This will leverage the Docker cache speeding up the following build when updating the source code files.[12]

Interestingly, developers sometimes need to avoid caching to improve performance (34 occurrences). Indeed, while caching at the Docker level is positive (*e.g.,* the previously-mentioned layer caching mechanism), using the cache of the package managers during the build is mostly negative. Such a mechanism, indeed, increases the image size (the cache is inside the resulting Docker image) but it does not speed up the build since those files will be discarded in the next build (or the layer caching mechanism will take over whatsoever). This is why developers tend to explicitly use options to avoid caching in `apk`, `pip` and `bundle`.

Finally, developers apply more specific changes to enable the use of caching in a few cases (8 occurrences). For example, we found that, in some cases (3 occurrences), developers adopt an external `VOLUME` with the dependency cache and mount it during the build to speed it up.

[12] https://github.com/detiuaveiro/social-network-mining/commit/020d5c3



```diff
@@ -1,9 +1,19 @@
 FROM python:latest
 
- COPY . /
+ COPY requirements.txt /
 
 RUN pip install -r requirements.txt
 
+ COPY ./Mongo /Mongo
+ COPY ./Postgres /Postgres
+ COPY ./Neo4j /Neo4j
+ COPY ./Enums /Enums
+ COPY ./ElasticSearch /ElasticSearch
+ COPY send.py /
+ COPY settings.py /
+ COPY api.py /
+
+
 EXPOSE 5000
 
 CMD [ "python", "./api.py" ]
```

Fig. 7 Example of a "Caching" change in leveraging the layer caching to optimize the build

4.1.4 Tweaks

The changes occurring less frequently are those aimed at optimizing the usage of tools (*Tool-specific*, 10 occurrences) and *Install Operations* (37 occurrences). An example for the first is switching to a more efficient tool (*i.e.,* from `npm` to `yarn`). This change helps to reduce the build time.[13] For the latter, developers usually build a part of the required binaries outside the container, to reduce the total build time of the image (10 occurrences). Also, they compile sources optimizing the targets and versions (6 occurrences) to reduce build time overhead and storage size. We report an example for *Avoid Dependency Upgrade* in Fig. 8, in which the flag `--keep-outdated` prevents the upgrade of `pip` packages before installing.[14]

🔍 **Results Summary**: Four main type of changes are performed by developers to improve the image size and build time of images: Those for *Debloating* (49%), changing the *Dockerfile Architecture* (36%), optimizing *Caching* (12%), and installation *Tweaks* (4%).

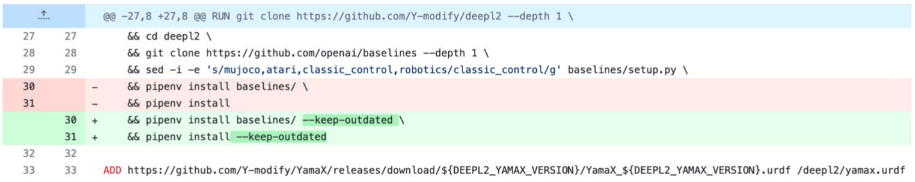
4.2 RQ$_2$: Measuring the Impact of Performance Changes

We report the results of the analysis conducted for RQ$_2$ in Table 2 (image size) and Table 3 (build time).

As for the image size, the most effective strategies are those involving the modification of the base image. In detail, modifying the variant of the base image led to a reduction in the size

[13] https://github.com/pnpm/benchmarks-of-javascript-package-managers

[14] https://github.com/Y-modify/deepl2-infra/commit/0edee8b



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

  ADD https://github.com/Y-modify/YamaX/releases/download/${DEEPL2_YAMAX_VERSION}/YamaX_${DEEPL2_YAMAX_VERSION}.urdf /deepl2/yamax.urdf
```

Fig. 8 Example of a "Tweaks" change avoiding the upgrade of the dependencies installed using `pip`

of the final image in most of the cases analyzed. This category showed a statistically significant effect with a large effect size, indicating its practical relevance. On average, changing the base image variant reduced the image size by 62% (69% when considering only the ones that actually resulted in reduced image size). Choosing a completely different base image led to substantial reductions in size as well, with statistical significance and a large Cliff's delta. In this case, however, the reduction is slightly more moderate (∼15%, 54% when considering only the cases that resulted in reduced size). Multi-stage builds also resulted in consistent and substantial reductions of the image size: When implementing such a pattern, the image size was reduced by 62%, on average (74% when considering only the ones that reduced image size).

Note that all the previously-reported patterns require a non-negligible rework of the Dockerfile. However, even simpler modifications, such as joining `RUN` instructions, result in significant and substantial improvements. On average, the image size gets reduced by ∼24% (30% when considering only the instances that reduced the image size).

All the other practices we analyzed, such as cleaning package manager cache or removing unused files, while generally beneficial, had more limited impact. Note that this is not only due to the low number of instances we analyzed (which probably led to non-significant differences), but also to the low gain obtained from such cases. Some of such practices do not change the image size at all (*e.g.,* the use of `clean` and `autoclean`) or cause very limited benefits (*e.g.,* removing the apk cache only reduces the image size by ∼2%).

As for build time, the results are more nuanced. First, changing the base image or its variant often causes an enormous increase in build time (131% and 156%, respectively). This increase in build time is statistically significant for the former category. The same is true for the adoption of multi-stage builds. Such changes implicitly require developers to choose a new base image for the last stage, which contains the actual image that will be generated. Again, using a smaller image (like developers often do) can require to introduce new `RUN` instructions to install dependencies that used to be included in the base image of the

Table 2 Impact of different change patterns on the image size

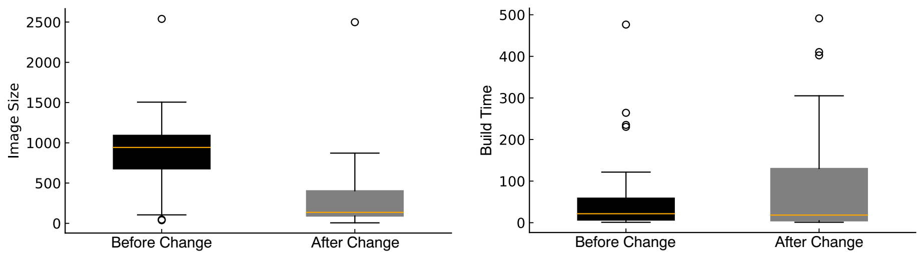
Category	Reduced	Neutral	Increased	p-value	*Cliff's δ*
change base image variant	25/26	0/26	1/26	<0.01	0.92 (large)
choose a different base image	8/12	1/12	3/12	0.18	
use multi-stage build	11/13	0/13	2/13	<0.01	0.69 (large)
join RUNs	8/10	2/10	0/10	0.01	0.80 (large)
remove unused bin. and dep.	5/7	1/7	1/7	0.04	0.57 (large)
remove inst. dep. and src.	2/4	2/4	0/4	0.18	
remove apt-get lists	2/4	2/4	0/4	0.18	
apt-get clean-autoclean	0/3	3/3	0/3	-	-
remove apk cache	2/2	0/2	0/2	0.50	
apk –no-cache	1/2	0/2	1/2	1.00	

Table 3 Impact of different change patterns on the build time

Category	Reduced	Neutral	Increased	p-value	*Cliff's δ*
change base image variant	6/26	5/26	15/26	0.01	-0.46 (medium)
choose a different base image	3/12	3/12	6/12	0.57	
use multi-stage build	5/13	2/13	6/13	0.54	
join RUNs	5/10	1/10	4/10	0.69	
remove unused bin. and dep.	2/7	3/7	2/7	0.81	
remove inst. dep. and src.	3/4	1/4	0/4	0.13	
remove apt-get lists	0/4	4/4	0/4	0.38	
apt-get clean-autoclean	1/3	1/3	1/3	1.00	
apk –no-cache	0/2	1/2	1/2	0.50	
remove apk cache	1/2	1/2	0/2	1.00	

single-stage Dockerfile. We could not observe any change pattern that leads to statistically significant improvements in terms of build time. For example, the use of multi-stage builds, though beneficial for image size, did not consistently reduce build time in our measurements. Likewise, joining `RUN` instructions, while helping reduce the number of layers, had a negligible or inconsistent effect on build time. The categories related `apk ––no-cache` or purging package managers' caches, showed no clear benefit in terms of build time. There are two cases in which we observed a reduction of build time when removing the apk and apt cache. There is no theoretical reason why this should happen: Indeed, both modifications imply the execution of an additional instruction, *i.e.,* the build time should remain the same in the best case scenario. We suspect such results could be due to temporary issues with the server (*e.g.,* higher network traffic). In both cases, however, the results are not significant. The analysis highlights that performance improvements in Dockerfiles are more reliably achieved in terms of image size rather than build time.

An interesting observation is that the change of base image requires the evaluation of a trade-off between image size and build time. We depict in Fig. 9a and b the impact of changing the base image variant on size and time, respectively. It is clear that such a solution produces a substantial reduction in image size, but it also substantially increases the build time.



(a) Image size before and after changing the base image.

(b) Build time before and after changing the base image.

Fig. 9 Effect of changing the base image on image size and build time

Results Summary: Changes that involve the base image are the most effective in reducing Docker image size, with statistically significant improvements and large effect sizes. Multi-stage builds and instruction joining (*e.g.,* `RUN`) also reduce image size effectively. On the other hand, improvements in build time are less consistent.

5 Discussion

In this section, we provide some takeaways extracted from our results and implications for future studies.

5.1 Takeaways

Finding 1. Choosing an efficient base image is key. Choosing a better *base image* in terms of size or embedded dependencies appears to be the most impacting change to improve performance. In fact, it is actually the most frequent change performed by developers (271 occurrences), and it allows to achieve a significant reduction of the final image size and build time. Previous work (Rosa et al. 2023) suggest that a rule of thumb could be to rely on *official* Docker images, better if they already contain some of the required dependencies. Analyzing more in detail the changes collected in RQ_1, developers usually switch to the *alpine* variant, or, in general, they switch to *alpine* base images.

Finding 2. There is some free lunch. As previously reported, choosing a good base image is fundamental. However, changing the base image is a double-edged sword. Reducing the base image size might result in significantly higher build time due to the overhead related to the installation of additional packages not ready-available in it. Having a smaller image is generally more desirable since it reduces the cost of deployment, but higher build times negatively impact the time to deploy. The problem here is that pre-defined base images are one size fits all that are rarely good enough as they are for any software system. There is, however, a solution to this problem that would allow developers to achieve both goals (*i.e.,* reducing both image size and build time). In our mining study, we found a very limited number of examples of cases in which developers extract a Dockerfile that they use as a tailored base image for the main Dockerfile. This allows them to have a small image size (they can use a small base image on which they can install all the required dependencies once) and lower build time (the Dockerfile dedicated to build the base image needs to be built less frequently than the main one). While this change increases the complexity of the project, it might result in optimal performance.

Finding 3. The cost of the change can be more than the improvement. Some changes provide a very small improvement that is less than the cost to perform it. An interesting case is commit[15] in which the removal of build and install dependencies increases the build size, as previously reported. Each added `RUN` instruction produces a layer causing a space wastage, nullifying the improvement of the change. In general, we observed that some changes (such as removing the apt-get lists or apk cache) result in negligible benefits both in terms of image size and build time. We recommend practitioners to pay less attention to such finer-grained optimization and focus on more impacting aspects before (such as, again, the base image).

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

Table 4 Summary table of the identified changes that are in overlap with existing catalogs, *i.e.,* Docker Official guidelines[a] (Off), *hadolint* tool (hadolint 2015) (Ha), Binnacle (Henkel et al. 2020) (Bi), DRIVE (Zhou et al. 2022) (DR), DOCKERCLEANER (Bui et al. 2023) (DOC), and the study of Ksontini et al. (2021) (Ks)

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

We reported the cases in which there is a partial (■) or full (✓) overlap
[a]Best practices for writing dockerfiles. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ (2023). [Online; Accessed 2 Jun 2022].

Finding 4. Some changes are useful only for specific usage patterns. As reported in the previous section, changes like *caching dependencies* or *copy/install requirements beforehand* are not effective in the first build of the image. However, they became effective only in successive build (*e.g.,* by caching maven dependencies). This pattern is positive when Dockerfiles are locally used for development, for which it is required to frequently run a build to test the containerized product. However, the contrary is true when they are integrated in CI/CD pipelines that do not rely on caching.

5.2 Implications

A part of the changes reported in our taxonomy (Fig. 4) are related to the best writing practices suggested by Docker.[16] Examples are *using multi-stage build* and *join* `RUN`*s*. Moreover, the existing catalogs of writing violations (*i.e.,* Dockerfile smells) cover also a part of those changes (hadolint 2015; Henkel et al. 2020). We report in Table 4 the changes identified in

[16] Best practices for writing dockerfiles. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ (2023). [Online; Accessed 2 Jun 2022].

our study that were already defined (entirely or partially) in previous work. Our investigation proposes a total of 25 new practices. In some cases, the practice is similar to an existing one but extended to a different tool or platform. An example is the usage of `--progress` flag with `npm`, which is conceptually similar to rule `DL3047` from *hadolint*. It is worth noting that some of the practices we identified and not present in any previous catalog, like the change of the base image variant, are very frequently adopted by developers and have a substantial impact on either build time or image size.

Our results provide clear indications for researchers and tool builders on the aspects they should focus on. In particular, future research should focus on approaches aimed at suggesting the modifications that require particular effort by developers to apply. An example can be the definition of approaches for the automated refactoring of complex Dockerfiles as multi-stage builds. Also, approaches that can suggest what are the unnecessary dependencies and sources that can be removed, taking as input a Dockerfile. Last but not least, recommending developers a better base image replacement can have a high impact on performance improvement. In this direction, existing approaches for base image recommendation could be adapted to this specific aim (Kitajima and Sekiguchi 2020; Zhang et al. 2022).

6 Threats to Validity

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

Construct Validity We assumed that the commits selected by our NLP filter have an effective impact in terms of build time and image size on the resulting Docker images. However, this could lead to false positives where the commit message reports the intention of improvement, but the change itself, by design, does not provide it. An example is the commit[17] where the type of change is not effective in reducing the build time of the image. We mitigate this by performing a manual validation on the changes selected by the filter. Moreover, our filtering approach is designed to be simple and explainable, *i.e.,* replying on a keyword-based marching heuristic, to mitigate any sampling bias.

Internal Validity There is a possible subjectiveness introduced during the manual annotation of the change applied by each commit. We mitigated this by adopting a conservative approach, *i.e.,* we did not "interpret" the commit change, but we mainly relied on information provided by the commit message. Also, the process has been executed independently by two different annotators discussing and resolving conflicting tags with a third annotator. Another threat to internal validity is the relatively low number of corresponding commits (11k out of 11.5M). Although this number may seem small, we conjecture it reflects the fact that performance improvements are rarely described explicitly in commit messages. On the other hand, despite we defined our queries from a large vocabulary analysis, some relevant but implicitly described changes may have been excluded. To answer RQ_2 we measured build time, which is an inherently stochastic variable. Docker build time depends on several factors, including the server load and the network traffic. To minimize the influence of chance on such measurement, we used an ad-hoc virtual machine on which no other processes were running. Besides, we repeated each build 20 times. It is still possible that the categories for which we observed a low number of significant improvements do have an impact on build time, but it is too small to be measured with our experiment (*e.g.,* a higher number of

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

builds would have been necessary). The reported Cohen's Kappa reflects some disagreement among annotators, mainly due to the fact that some tags are very similar and might be used interchangebly, given the fact that we did not establish the specific tags before starting the manual labeling phase (finding out the categories was part of the goal of our mining study). For example, a commit[18] from *amitmbee/krapp* involved a change to the source repository of the base image: The developer changed the base image from `alpine-node:latest` to `mhart/alpine-node:latest`. One of the evaluators tagged this change as a simple *change base image*, while the other used the *change base image variant* tag, which is more specific. Note that, in this case, both tags might be correct, but it is important to tag similar changes in a consistent way. To mitigate this threat and ensure consistent labeling, a third annotator was involved to resolve such conflicts.

External Validity The taxonomy proposed in our study is based mainly on open-source Dockerfiles. This means that there could be some differences when applied in an industrial context. In addition, since our dataset was built by filtering commits based on performance-related keywords in commit messages, there is a potential threat that less common or implicitly applied optimization strategies are underrepresented. As a result, the taxonomy may emphasize more frequently reported or explicitly documented practices. Currently, we considered only files which name, or part of it, contains the string "dockerfile" (not case sensitive). We may miss some Dockerfiles that are not named according to the convention. This introduces a potential threat to validity, as some Dockerfiles may not be included in our analysis. However, we believe that this is a negligible issue, as the vast majority of Dockerfiles are named according to the convention. However, the practices that are captured in our taxonomy cover some of those suggested in the official Docker guidelines[19] and as code smells (hadolint 2015). This means that our finding are an enrichment of the existing practices used by developers. Finally, we could measure the impact on build time and image size (RQ_2) only for a small number of categories of changes, and we needed to exclude several of them because we did not have enough data points. It is possible that some of the categories with fewer changes we could not test in this study have a bigger impact than the ones we focused on.

7 Conclusion and Future Work

Optimizing the resources used by Dockerfiles and Docker images is one of the most important aspects in which developers invest their effort. In this paper, we presented an in-depth empirical evaluation of the changes performed by developers in the open-source aimed at improving the image size and build time of Docker images. First, we extracted a set of improvement changes from git repositories, filtering them by combining NLP techniques and manual validation, to exclude false positives. Then, we annotated the performed changes to reduce build time and image size, to finally group them in a taxonomy of changes. While our results provide a view of performance-oriented changes in Dockerfiles, some limitations remain. Our taxonomy is based on a filtered and curated subset of open-source commits, which may not fully represent industrial practices or all possible optimizations. Additionally, the impact of certain changes-such as caching-related ones-can vary depending on the environment in

[18] https://github.com/amitmbee/krapp/commit/7c80f82

[19] Best practices for writing dockerfiles. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ (2023). [Online; Accessed 2 Jun 2022].

which Docker images are built and used (*e.g.,* CI/CD pipelines vs. local development). These factors should be taken into account when generalizing or applying our findings in practice. As a future direction, we plan to run a dedicated study on the impact of the changes we found on both image size and build time. We also plan to investigate what effort is needed to perform such changes to refine our catalog in a cost-effective perspective.

Author Contributions Giovanni Rosa, Emanuela Guglielmi, Mattia Iannone, Simone Scalabrino, and Rocco Oliveto contributed to the study conception and design. Material preparation, data collection and analysis were performed by Giovanni Rosa, Emanuela Guglielmi, and Mattia Iannone. The first draft of the manuscript was written by Giovanni Rosa and all authors reviewed and edited the final manuscript. All authors read and approved the manuscript.

Funding This work was supported by the Italian Government (Ministero della Università e della Ricerca, PRIN 2022 PNRR) under the project "RECHARGE: Monitoring, Testing, and Characterization of Performance Regressions," grant n. P2022SELA7, funded by European Union - NextGenerationEU.

Data Availability To make our results verifiable and replicable, we provide a publicly available replication package (Rosa et al. 2024), which contains the results of the tagging and the experimental measurements we acquired.

Declarations

Conflict of Interest The authors declare that they have no conflict of interest.

Ethical Approval Not applicable.

Informed Consent Not applicable.

Clinical Trial Number Not applicable.

References

Azuma H, Matsumoto S, Kamei Y, Kusumoto S (2022) An empirical study on self-admitted technical debt in dockerfiles. Empir Softw Eng 27(2):1–26

Bernstein D (2014) Containers and cloud: from lxc to docker to kubernetes. IEEE Cloud Comput 1(3):81–84

Bui QC, Laukötter M, Scandariato R (2023) Dockercleaner: Automatic repair of security smells in dockerfiles. In: 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), p To Appear. IEEE

Cito J, Schermann G, Wittern JE, Leitner P, Zumberi S, Gall HC (2017) An empirical analysis of the docker container ecosystem on github. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp 323–333. IEEE

Durieux T (2024) Empirical study of the docker smells impact on the image size pp 1–12

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

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

Henkel J, Bird C, Lahiri SK, Reps T (2020) A dataset of Dockerfiles. In: Proceedings of the 17th international conference on mining software repositories, pp 528–532

Henkel J, Bird C, Lahiri SK, Reps T (2020) Learning from, understanding, and supporting devops artifacts for docker. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), pp 38–49. IEEE

Henkel J, Silva D, Teixeira L, d'Amorim M, Reps T (2021) Shipwright: a human-in-the-loop system for dockerfile repair. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp 1148–1160. IEEE

Jiang Q (2022) Improving performance of docker instance via image reconstruction. In: International conference on big data intelligence and computing, pp 511–522. Springer
Kitajima S, Sekiguchi A (2020) Latest image recommendation method for automatic base image update in dockerfile. In: International conference on service-oriented computing, pp 547–562. Springer
Ksontini E, Kessentini M, Ferreira TdN, Hassan F (2021) Refactorings and technical debt in docker projects: an empirical study. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 781–791. IEEE
Landis JR, Koch GG (1977) An application of hierarchical kappa-type statistics in the assessment of majority agreement among multiple observers. Biometrics 363–374
Lin C, Nadi S, Khazaei H (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), pp 371–381. IEEE
Macbeth G, Razumiejczyk E, Ledesma RD (2011) Cliff's delta calculator: a non-parametric effect size program for two groups of observations. Univ Psychol 10(2):545–555
Rastogi V, Davidson D, De Carli L, Jha S, McDaniel P (2017a) Cimplifier: automatically debloating containers. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, pp 476–486
Rastogi V, Niddodi C, Mohan S, Jha S (2017b) New directions for container debloating. In: Proceedings of the 2017 workshop on forming an ecosystem around software transformation, pp 51–56
Rosa G, Guglielmi E, Iannone M, Scalabrino S, Oliveto R (2024) Replication package for "mining and measuring the impact of change patterns for improving the size and build time of docker images". https://doi.org/10.6084/m9.figshare.24579580
Rosa G, Scalabrino S, Bavota G, Oliveto R (2023) What quality aspects influence the adoption of docker images? ACM Trans Softw Eng Methodol
Rosa G, Zappone F, Scalabrino S, Oliveto R (2024) Fixing dockerfile smells: an empirical study. Empir Softw Eng 29(5):108
Skourtis D, Rupprecht L, Tarasov V, Megiddo N (2019) Carving perfect layers out of docker images. In: 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)
Spencer D (2009) Card sorting: designing usable categories. Rosenfeld Media
Woolson RF (2007) Wilcoxon signed-rank test. Wiley encyclopedia of clinical trials, pp 1–3
Wu Y, Zhang Y, Wang T, Wang H (2020) Characterizing the occurrence of dockerfile smells in open-source software: an empirical study. IEEE Access 8:34127–34139
Zerouali A, Mens T, Decan A, Gonzalez-Barahona J, Robles G (2021) A multi-dimensional analysis of technical lag in Debian-based docker images. Empir Softw Eng 26(2):19
Zhang Y, Vasilescu B, Wang H, Filkov V (2018) One size does not fit all: an empirical study of containerized continuous deployment workflows. In: Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp 295–306
Zhang Y, Zhang Y, Mao X, Wu Y, Lin B, Wang S (2022) Recommending base image for docker containers based on deep configuration comprehension. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 449–453. IEEE
Zhou Y, Zhan W, Li Z, Han T, Chen T, Gall H (2022) Drive: Dockerfile rule mining and violation detection. arXiv:2212.05648