What Quality Aspects Influence the Adoption of Docker Images?

GIOVANNI ROSA, STAKE Lab, University of Molise, Italy
SIMONE SCALABRINO, STAKE Lab, University of Molise, Italy
GABRIELE BAVOTA, Software Institute, USI Università della Svizzera Italiana, Switzerland
ROCCO OLIVETO, STAKE Lab, University of Molise, Italy

Docker is a containerization technology that allows developers to ship software applications along with their dependencies in Docker images. Developers can extend existing images using them as base images when writing Dockerfiles. However, a lot of alternative functionally-equivalent base images are available. While many studies define and evaluate quality features that can be extracted from Docker artifacts, it is still unclear what are the criteria on which developers choose a base image over another.

In this paper, we aim to fill this gap. First, we conduct a literature review through which we define a taxonomy of quality features, identifying two main groups: *Configuration-related features* (*i.e.,* mainly related to the Dockerfile and image build process), and *externally observable features* (*i.e.,* what the Docker image users can observe). Second, we ran an empirical study considering the developers' preference for 2,441 Docker images in 1,911 open-source software projects. We want to understand (i) how the *externally observable features* influence the developers' preferences, and (ii) how they are related to the *configuration-related features*. Our results pave the way to the definition of a reliable quality measure for Docker artifacts, along with tools that support developers for a quality-aware development of them.

CCS Concepts: • **Software and its engineering → Software notations and tools**.

Additional Key Words and Phrases: Empirical software engineering, Software maintenance, Container virtualization, Docker

ACM Reference Format:
Giovanni Rosa, Simone Scalabrino, Gabriele Bavota, and Rocco Oliveto. 2018. What Quality Aspects Influence the Adoption of Docker Images?. In . ACM, New York, NY, USA, 29 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Deploying software and keeping it in operation is technically challenging. Moreover, the production environment is rarely identical to the development environment, which increases the risk of failures, *e.g.,* due to missing runtime dependencies, or even security vulnerabilities.

Containerization allows developers to ship software applications along with dependencies and the execution environment. Thanks to containerization, it is possible to run the application on any system [5] and test it in the same environment used in production. Docker[1] is one of the most popular containerization platforms used in the DevOps workflow. Docker allows releasing applications with their dependencies through containers (*i.e.,* virtual environments) sharing the kernel of the host operating system. The specification file of a Docker image is called Dockerfile. DockerHub[2]

[1]https://www.docker.com/
[2]https://hub.docker.com/

is an online repository similar to those for source code, *i.e.,* GitHub, which hosts a set of Docker images that can be downloaded and used by developers.

When writing the Dockerfile for a given application, developers usually start from a pre-existing image containing the basic dependencies needed. For example, to containerize a Java application, it will be necessary to provide the Java Runtime Environment (JRE): Therefore, a base image with the JRE could be adopted. However, many alternative images exist that provide the same (or analogous) dependencies, and developers find it difficult to search for Docker images on DockerHub [4, 15]. In general, we can extract different features from Docker images, *i.e., externally observable features*, that influence their adoption as they are what developers and image users can observe when they have to choose a Docker image to use. Such features include, for example, the image size [18] related to the resources that the image will use, and the presence of software vulnerabilities [24, 30, 39] which can lead security risks. Such *externally observable features* are influenced by *configuration-related features*, *i.e.,* mainly related to development aspects of the Dockerfiles that might positively or negatively affect the resulting Docker image. Examples are the presence of Dockerfile smells [20, 34] which can lead to the introduction of security issues [39]. Static analysis tools can support developers to follow best practices in Dockerfiles [1, 12, 36] and, thus, minimize the presence of *internal* quality issues. However, they may not be sufficient to assess the absence of code smells [23]. Despite such exemplary features and the presence of a plethora of studies that focus on specific quality issues, the literature lacks a general view of what are the externally observable and configuration-related features of Docker images and Dockerfiles. Similarly, to the best of our knowledge, it is unclear (i) how externally observable features impact developers' preferences when they have to choose a Docker image, and (ii) the impact of configuration-related features on the *external* ones. In this paper, we aim at filling these gaps. First, we reviewed 31 papers to define a comprehensive taxonomy of externally observable features and configuration-related features features of Docker images and Dockerfiles. Then, we conduct an empirical study on a dataset of 2,441 open-source Docker images. We aim at finding out what *external* features impact the developers' preferences in terms of actual *adoptions* (*i.e.,* how frequently they appear in the `FROM` statements of app-specific Dockerfiles) and perceived quality, intended as the *prominence* of a Docker image over others (*i.e.,* number of stars on DockerHub). Our results show that, as expected, official Docker images have a positive relationship with both *adoptions* and *prominence*. Besides, both image size and the number of exposed secrets (*i.e.,* a metric related to security) negatively impact the developers' preferences. Interestingly, the number of vulnerabilities only impacts the prominence of the image, but not the actual number of adoptions. This result suggests that developers are aware that some problems affect the quality of the images, but this does not change their behavior when they have to choose a Docker image to use (mostly because they are not aware of alternatives [15]). Moreover, our results show that the less the number of SLOC, the less the occurrence of vulnerabilities as also shown in previous studies [2, 25]. In the same way, also the image size decreases when the number of LOCs are decreasing. This means that a smaller image size has a positive impact on the developers' preferences. Also, we found no relationship between the presence of Dockerfile smells and any of the external features. Shell script smells, instead, have an impact on security-related features. However, there are some exceptions. This is mainly because, as we performed a correlation study, it can not be implied causality based on these results. For example, not all instructions (in terms of SLOC) directly impact the image size. For example, this not applies when removing instructions like `EXPOSE` or `LABEL`. On the other hand, shell script smells are not always related to security. It is proven that mature Docker images tend to have fewer security issues [30], despite the number of smells.

Fig. 1. An example of Dockerfile from the official documentation

```
FROM ubuntu:18.04

COPY . /app
RUN make /app
CMD ["python", "/app/app.py"]
```

To summarize, we provide the following contributions:

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

The rest of our paper is organized as follows. In Section 2 we give some preliminary concepts to better understand how Docker works, and also we discuss some of the relevant works from the literature. In Section 3 we describe the procedure used for building the taxonomy of features and metrics of Dockerfile artifacts. In Section 4 we present some hypotheses related to the impact of the quality features on developers' preferences. In Section 5, we present our empirical study to evaluate the impact of the quality features on the developers' preferences. We discuss the results in Section 6, and the threats of validity in Section 7. Finally, in Section 8 we provide the conclusion along with future directions.

2 BACKGROUND AND RELATED WORK

In this section, we describe some preliminary concepts about Docker, its functionalities, and the tools typically used to assess the quality of Docker artifacts.

2.1 Docker Basics

Docker is one of the most popular containerization technologies. The main purpose is to ship an application along with its dependencies and execution environment. A Dockerfile is the specification file behind a Docker application, in which there are source code lines that define the packages and dependencies needed by the application, in addition to the configuration of the environment. An example of a Dockerfile is reported in Fig. 1.

The programming language used to define a Dockerfile is composed of specific instructions[3]. Each Docker instruction performs specific actions, usually defined by shell script code. For example, the main Docker instruction with which each Dockerfile begins is `FROM`, which defines a so-called *base image* from which the new Docker image defined in the Dockerfile can inherit dependencies and configurations. Every Docker image can be used as *base image* and, therefore, extended. The `RUN` instruction contains one or more commands that will be executed in a shell environment (*i.e.,* `RUN <command>`), that is by default `/bin/sh -c`. Starting from a Dockerfile, a Docker image containing the application is created via the `build` operation. While building the image, Docker runs all the instructions in the Dockerfile (*e.g.,* to download dependencies and resources or to build the software product). The Docker image is then ready to be used to

[3] https://docs.docker.com/engine/reference/builder/

execute the application. Each image is composed of *layers*, which are snapshots of the image during the build process. Each layer is created by a Docker instruction that makes changes to the image itself. The main purpose is to make the build process modular and to speed it up using caching: Instead of running all the instructions of a Dockerfile, it is possible to save time and resources by re-using pre-built layers, when possible (*e.g.,* avoid re-installing of software packages). A Docker image is executed in a *container*, *i.e.,* a lightweight virtual machine that as their own resources, such as networks and storage volumes.

Each Docker image is uniquely identified by the *digest*, a hash value computed at build time based on the composition of the image. However, it is common practice to assign a meaningful name (*i.e.,* a *tag*) to the images, so that it is possible to refer to them more easily. The image *tag* is usually composed of the name of the software installed in it (*e.g.,* php), its version (*e.g.,* 7.0), and its *flavor* (*e.g.,* slim).[4] The latter might denote differences in terms of non-functional requirements (*e.g.,* the reduced size). An example of a tag is `"name:version-flavor"`. It is worth specifying that the same Docker image can have multiple tags, thus the only way to identify unique images is using the *digest*.

Similarly to software dependency management systems (such as Maven), all the Docker images are distributed through *registries*, from which developers can retrieve and use them. There are two kinds of registries: private and public. Private registries are usually restricted to specific companies or usages (*e.g.,* an internal registry of a large software system to host and deploy images on Kubernetes), while the main public registry is DockerHub[5]. There are four types of images hosted on DockerHub. First, we have official images[6], maintained following the official images review guidelines[7]. The aim is to ensure the overall quality of such images. Second, there are images from verified publishers, *i.e.,* publishers that can be trusted, but that do not necessarily produce official images that follow the previously mentioned guidelines. Third, we have images that are part of the Docker Open Source program[8], maintained by organizations that are members of that program. Last, we have the non-official images, which are provided by the users of the Docker community.

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

2.2 Support tools for Docker Images and Dockerfiles

Several tools are available for Docker images and Dockerfiles to support developers during development. The most used is *hadolint* [1], a static analysis tool to check best practices[9] in Dockerfiles. The tool parses the Dockerfile into an equivalent AST and verifies a set of rules. Each rule, defined by an identifier, is associated with a writing best practice. For example, the rule `DL3008` checks for missing version pinning for packages installed via `apt-get`. The number of rule violations is a common measure of the quality of Dockerfiles [5]. Other tools, instead, assess the security of

[4] https://docs.docker.com/engine/reference/commandline/tag/
[5] https://hub.docker.com/
[6] https://docs.docker.com/docker-hub/official_images/
[7] https://github.com/docker-library/official-images#review-guidelines
[8] https://www.docker.com/community/open-source/application/
[9] https://docs.docker.com/develop/develop-images/dockerfile_best-practices/

Docker images. For example, *docker-bench-security*[10] is a tool that checks for various security best practices for the deployment of Docker applications in production environments[11]. Moreover, there is a built-in tool for vulnerability scan on Docker images, *i.e.,* the command *docker scan*[12], that checks for Common Vulnerabilities and Exposures[13] (CVE). Unfortunately, the tool requires a premium plan for an unlimited number of scans. An open-source alternative is the *clair-scanner* tool[14], which also performs checks for the presence of CVEs on Docker images. Furthermore, some tools allow performing reverse engineering on Docker images to extract the source code that created each layer. An example is a tool *whaler*[15], which besides the source code, also extracts additional information such as the main user account, the environment variables, and if there are exposed secrets inside the Docker image (*i.e.,* sensitive information such as login credentials). In our study, to extract the quality features of Docker images and Dockerfile, we adopt *hadolint* to detect smells, *clair-scanner* for security vulnerabilities, and *whaler* to extract the additional information from Docker images.

2.3 Studies on the quality of Docker Artifacts

SSeveral studies analyzed the quality aspects of Docker images and Dockerfiles.

Wu *et al.* [34] conducted an empirical analysis on the occurrence of Dockerfile smells, involving a large-scale dataset of Dockerfiles. Their findings show that smells are very common in Dockerfiles, as they are present in 84% of analyzed GitHub projects. Also, the number of smells is related to the programming language used. Moreover, popular and young project repositories and projects with many contributors tend to have fewer Dockerfile smells. We considered in our study some of the dependent and independent variables involved in their study as metrics to include in our taxonomy. Then, we used those metrics to extract the measured features from the Docker images involved in our empirical study. Their analysis is mainly focused on the quality assessment of Dockerfiles in terms of the occurrence of smells, while, in our study, we extend the concept of quality to both external and configuration features that can be measured on Docker images and Dockerfiles.

Zhang *et al.* [41] performed an empirical study on the impact of the evolutionary trajectories of Dockerfiles. The evolutionary trajectories describe the frequency and type of modifications performed by the Dockerfile project maintainers. Then, through a regression analysis, the authors evaluate their impact on the quality and image build latency. The results show that different types of evolutionary categories have a different impact on quality. In our study, we do not consider the change history of the Dockerfiles, but it is useful to evaluate the independent variables analyzed in their study. However, we do not consider evolutionary trajectories, because they are correlated with the quantity of best practices violations that we also consider.

Ksontini *et al.* [19] performed a study on refactoring operations and technical debts in open-source Docker projects. As a result, they propose a taxonomy of refactoring operations, where the most applied are those reducing the size of Docker images and improving the extensibility of docker-compose specification files. Also, a set of technical debts is defined. The main difference with our taxonomy is that we propose a set of specific metrics and features measuring the quality perceived by developers for Docker images and Dockerfiles. Moreover, we considered in our taxonomy the features related to refactoring operations and technical debts, that are related to the quality, involved in their study.

[10] https://github.com/docker/docker-bench-security
[11] https://www.cisecurity.org/benchmark/docker
[12] https://docs.snyk.io/more-info/getting-started/snyk-integrations/docker/scanning-with-the-docker-cli
[13] https://cve.mitre.org/
[14] https://github.com/arminc/clair-scanner
[15] https://github.com/P3GLEG/Whaler

Azuma *et al.* [3] conducted a study where they categorize self-admitted technical debts (SATDs) in Dockerfiles. As a result, they proposed a classification identifying five classes and eleven subclasses of different Docker SATDs. Also, code debt and test debt are common SATDs in Dockerfiles, where 42% of them are Docker-specific. The main difference with our study is that SATDs are related only to Dockerfiles, whereas we also consider Docker images. Moreover, not all the SATDs are related to code quality, but also to different non-functional aspects (*e.g.,* design, testing, and maintainability). We only included in our taxonomy only the aspects related to SATDs that can influence the quality of Dockerfiles.

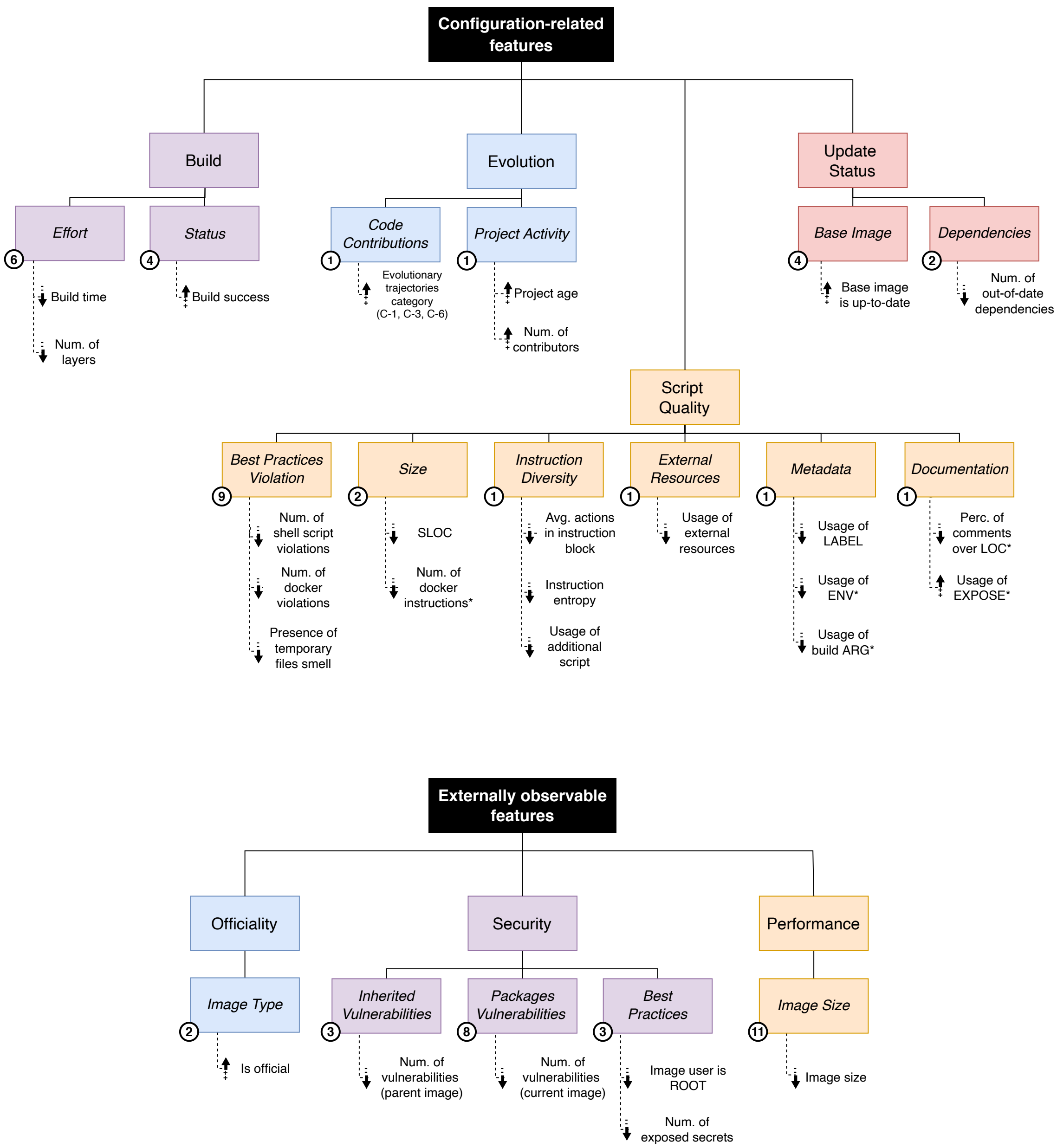
Ibrahim *et al.* [15] conducted an empirical study to evaluate the differences among Docker images hosted on DockerHub to support users to select the most suitable image to be adopted. Their results show that official images are more popular than community images. They show that community images are more resource-efficient than the studied software systems. Also, there are fewer security vulnerabilities than in their respective official images. In our study, we evaluate the adoption of Docker images instead of popularity. In addition, we analyze a larger set of features extracted from Docker images and Dockerfiles, defining also a detailed taxonomy of these features.

However, none of the previous studies evaluate the perspective of the developers and image users. The results of our work provide the missing piece in terms of how the presence of smells [35] and other internal quality issues related to the Dockerfiles [3, 19] impact on the adoption of Docker images. Moreover, our results are complementary to those of Ibrahim *et al.* [15], providing a different perspective regarding the actual usages of the Docker images, considering at the same time also the prominence of a specific Docker image over the others by taking into account the stargazers count.

Table 1. Inclusion and exclusion criteria for the selection of primary studies.

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

Fig. 2. Taxonomy of external and configuration features and metrics. For each feature, the number of references from the literature is reported. For each metric, an up or down arrow indicates if it is positively or negatively correlated to the feature it measures.



3 DISCOVERING EXTERNAL AND CONFIGURATION FEATURES OF DOCKER ARTIFACTS

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

3.1 Methodology

The *goal* of this preliminary study is to collect a set of configuration-related features and externally observable features of Docker images and Dockerfiles. To achieve this, we conduct a literature review of scientific articles about Docker quality, and we qualitatively analyze them to extract the information related to features and metrics. We have not performed a rigorous Systematic Literature Review (SLR) on quality aspects because the topic is too broad and it would have been outside the scope of this step (*i.e.,* selecting quality metrics). We describe below, in detail, the procedure we followed.

3.1.1 Identification of Relevant Articles. We searched for studies regarding Docker quality, as a general topic. To do this, we relied on Google Scholar, and we used the generic query *"docker quality"*. We collected a core set of articles that conduct studies on the quality of Docker images and Dockerfiles. Specifically, starting from the first paper returned by Google Scholar, we considered all subsequent papers stopping when the title and abstract did not contain the keywords *docker* and *quality* (∼ 30 results). We defined a set of inclusion and exclusion criteria, reported in Table 1, for selecting the articles of interest. After having collected the first set of papers, we read their titles and abstracts, and we verified the criteria *IC1*, *IC2*, *IC3*, *EC1*, *EC2*, *EC5*. At this stage, if we were not sure whether any of the used criteria were met, we kept the paper. Next, we used snowballing (*i.e.,* we analyzed the relevant references of the selected papers) and looked for more recent papers citing them by relying, again, on Google Scholar. We used the previously described process to filter them and include, in the end, only the possibly relevant ones. We applied a less strict filter on the title and abstract, also looking for words related to quality improvements (*e.g.,* refactoring, technical debt, repair) or quality-related aspects (*e.g.,* smells, build failures, security, performance, bugs). Finally, we carefully read the whole papers and filtered them using all the inclusion and exclusion criteria. In total, we analyzed 75 articles. We excluded 44 of them, and we were left with a total of 31 relevant articles to analyze in the next steps.

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

3.1.2 Qualitative Analysis Methodology. We analyzed the selected articles to find out the discussed metrics and features related to quality from the literature. For extracting the information of interest, we adopted the card sorting approach [32]. We identified, for each paper, the quality *features* and the possible *metrics* defined to measure them. Two of the authors, independently, assigned one or more tags to each article by distinguishing tags related to the quality *features* and the ones related to the quality *metrics*. Given the set of assigned tags for each category (*features* and *metrics*), we analyzed them, aiming at using a unique expression when the two evaluators used different tags for expressing the same concept

Table 2. Summary table of Docker configuration-related and externally observable metrics. The symbol * means that is a newly introduced metric.

	Metric	Rationale	How to compute	Reference
Configuration-related features	*Build time*	Build time of the Docker image	Build time measured from the execution time of the `docker build` command	[12, 14, 19, 20, 41, 42]
	Num. of layers	Build effort of the Docker image	Number of layers that compose the Docker image, using `docker history` command	[41]
	Build success	Build status	`TRUE` if the build was successful completed, `FALSE` o/w	[5, 12, 13, 35]
	Evolutionary trajectories category	Evolution of Dockerfile over time	Clustering of Dockerfile evolutionary vectors, computed using changes history [40]	[41]
	Project age	Evolution/Project activity	Time passed between first and last commit, measured in seconds	[34]
	Num. of contributors	Evolution/Project activity	Number of contributors that made at least one commit to the project repository	[34]
	Num. of shell script smells	Presence of code smells	Number of Docker-related violations (DL-XXXX) detected by *hadolint* tool	[3, 8, 11–13, 19, 20, 34, 41]
	Num. of docker smells	Presence of code smells	Number of Shell-related violations (SC-XXXX) detected by *hadolint* tool	[3, 8, 11–13, 19, 20, 34, 41]
	Presence of temporary files smell	Presence of code smells	Semi-automatic approach as described by Lu *et al.* [23]	[23]
	SLOC	Size of the Dockerfile	Number of lines of code (LOC), excluding code comments	[34, 41]
	Num. of docker instructions	Dockerfile complexity	Total number of the Docker instructions (*e.g.*, `RUN`, `COPY`, etc.) used in the Dockerfile	[34, 41]*
	Layer size	Dockerfile complexity	Average number of concatenated commands per instruction block (separated by &&)	[41]
	Instructions entropy	Dockerfile complexity	Shannon entropy computed among the different (unique) Docker instruction used in the Dockerfile	[41]
	Usage of additional script	Usage of additional resources	`TRUE` if the Dockerfile contains a shell script execution (*e.g.*, running bash scripts ending with `.sh`), `FALSE` o/w	[41]
	Usage of external resources	Usage of additional resources	`TRUE` if `wget` or `curl` commands calling external URLs are used in the Dockerfile, `FALSE` o/w	[41]
	Usage of LABEL	Usage of instructions for image metadata	`TRUE` if the `LABEL` instruction is used in the Dockerfile, `FALSE` o/w	[41]
	Usage of ENV	Usage of instructions for image configuration	`TRUE` if the `ENG` instruction is used in the Dockerfile, `FALSE` o/w	[41]*
	Usage of build ARG	Usage of instructions for image configuration	`TRUE` if the `ARG` instruction is used in the Dockerfile, `FALSE` o/w	[41]*
	Perc. of comments over LOC	Presence of code documentation	Percentage of number of comments over SLOC, computed as: *n. comments/SLOC*	[41]*
	Usage of EXPOSE	Presence of code documentation	`TRUE` if the `EXPOSE` instruction is used in the Dockerfile, `FALSE` o/w	[41]*
	Is base image up-to-date	Update status	Checking the Docker image repository on DockerHub for image tag updates	[16, 19, 20, 30]
	Num. of out-of-date dependencies	Update status	Checking the software packages repository for the presence of updates running `apt update` (for Debian-based images)	[38, 39]
Externally observable features	*Is official*	Vendor of the Docker image	`TRUE` if the Docker image has the *Official Image* badge in *DockerHub*	[10, 34, 41]
	Image size	Size of the Docker image	Image size measured in bytes, computed by adding up the size of all the image layers obtained from the `docker history` command	[3, 12, 19, 20, 23, 25, 31, 33, 36, 41–43]
	Num. of vulnerabilities (parent image)	Security vulnerabilities	Number of security vulnerabilities (v) considering only the base image (BI) using *Clair* scanner (BI_v)	[21, 25, 30]
	Num. of vulnerabilities (current image)	Security vulnerabilities	Number of security vulnerabilities (v) introduced by packages added in the resulting image (RI), using *Clair* scanner ($RI_v - BI_v$)	[15, 16, 20, 24, 33, 37–39]
	Image user is root	Security best practices	`TRUE` if the Docker image runs as root-enabled container, `FALSE` o/w (*e.g.*, using the *whaler* tool)	[21, 24]
	Num. of exposed secrets	Security best practices	Number of exposed login credentials or access tokens detected in the Docker image (*e.g.*, using the *whaler* tool)	[30]

(*e.g.,* “image size” and “size”). The two evaluators discussed the cases in which there were conflicts on the assigned tags, aiming at reaching a consensus.

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

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

3.2 Taxonomy of Quality Features and Metrics

The resulting taxonomy is described in Fig. 2. The boxes with italicized text indicate the features, while the others indicate categories of features we introduced in the taxonomy. Also, in Table 2, we report the quality metrics and the papers resulting from the literature review. The numbers in the circular badges, instead, indicate the number of papers that use the feature. Next, we describe the categories we identified for both configuration and external features.

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

Build. With this category, we indicate the aspects related to the build process of the Dockerfile. A slow build, for example, increases the time needed to update the software in production if continuous deployment is adopted. The *Effort* feature represents the resources involved in the build process (*e.g.,* time) [41], while the *Status* feature indicates the success or failure of the build process (*i.e.,* if Docker image builds or not) [35].

Evolution. This category embraces the aspects that are related to the evolution of the Dockerfile. The *Code Contributions* feature indicates the modifications made to the Dockerfile in time. The *Project Activity* feature, instead, describes the aspects related to the development process, such as team composition. Large development teams may be better at writing good quality Dockerfiles (*i.e.,* more technical knowledge) [34].

Script Quality. This category contains all the features strictly related to the quality of the source code. The feature *Violation of Best Practices* represents the presence of Dockerfile smells [34]. The feature *Dockerfile Size* represents the aspects related to the size of a Dockerfile, such as the number of lines of code. The *Instruction Diversity* feature is related to the homogeneity of the source code: A more heterogeneous code (*i.e.,* source code that has many different instructions) can lead to misleading developers [41]. The *External Resources* feature regards the usage of resources not provided in the original project repository, such as libraries or other files downloaded from remote servers [41]. The feature *Metadata* describes the use of meta-data in the Dockerfile, such as environment variables or the `LABEL` instruction [41]. Finally, the feature *Documentation* describes the use of documentation in the Dockerfile [41]. Code comments are an example of documentation. If the script quality of the Dockerfile is low, it is intuitively more likely that different kinds of issues arise (*e.g.,* security-related) given the lower maintainability [20, 34].

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

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

Officiality. With this first category, we indicate the degree of officiality of the image or of the developer(s) who published it. It is reasonable to assume that official images, or images published by trusted developers, are perceived better by developers because they are preferred over unofficial ones [10].

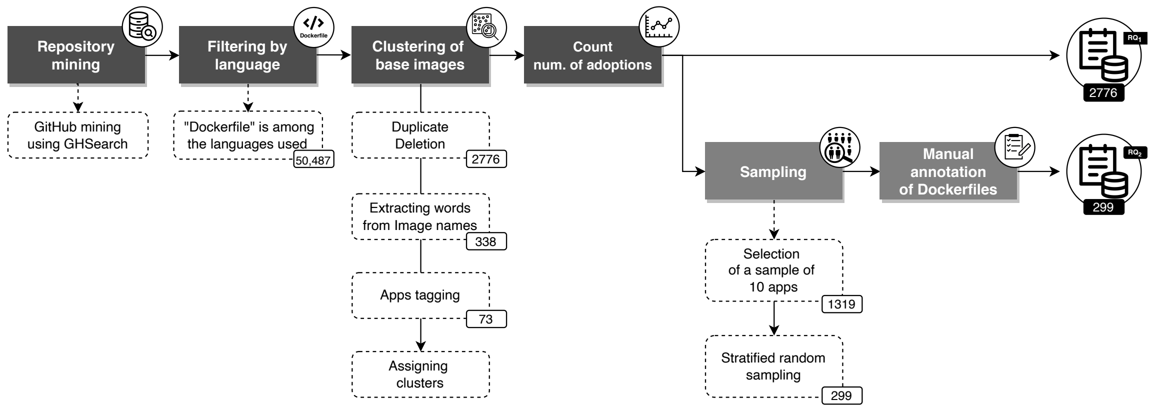
Performance. The way in which Docker images use the available resources might be crucial for developers since it also impacts the cost of operation. *Image Size*, specifically, is the only relevant feature related to this category, and it indicates the storage needed to use the image. Developers tend to dislike images bigger than necessary (*e.g.,* if they contain unnecessary software packages) [20].

Security. We include, in this last category, all the security-related aspects of a Docker image. The *Best Practice* feature concerns the adoption of the main security best practices of a Docker image. An example of best practice in terms of security is the usage of a user different from root, as the default user, when the image is executed. The *Inherited Vulnerabilities* and *Packages Vulnerabilities* features are related to the number of security vulnerabilities found in the image based on the *Common Vulnerabilities and Exposures* (CVE) database. The first one only concerns the parent image of the actual Docker image (*i.e.,* the base image used in the Dockerfile), while the second one concerns the additional software installed in the image. Developers must prefer images that provide all the necessary security-related features, to avoid security risks [21, 24].

3.2.3 Metrics. Table 2 describes in detail the metrics defined in our taxonomy and the features that they aim at capturing. While most of them were already defined in the papers we analyzed, we introduced some new metrics and variations of existing ones to better measure some of the features that compose our taxonomy. We describe below only the differences with respect to the existing ones, which are summarized in Table 2.

Configuration-related features. We introduced *Num. of docker instructions*, a new metric for measuring the *Size* of a Dockerfile. Such a metric counts the number of Docker instructions in the Dockerfile. Since each instruction of a Dockerfile will be converted to an image layer, a Dockerfile having many instructions will generally have a higher number of layers. It is worth noting that the number of instructions might be lower than the LOCs since a single instruction might encompass many lines. For the feature *Metadata*, we defined two more metrics: *Usage of ENV*, which measures the number of environment variables used in the Dockerfile, and *Usage of build ARG*, which measures the number of build arguments. Such metrics are inspired by *Usage of LABEL* [41], which indicates the presence of the LABEL instruction in Dockerfiles. The metrics *Perc. of comments over LOC* and *Usage of EXPOSE* are additional measures for the *Documentation* feature. The first one is a variation of a metric defined by Zhang *et al.* [41]: While the original version measures the absolute number of comments, our metric computes the percentage ratio between the number of comments and LOC. It is expected that the ratio, more than the absolute number of comments, is important to determine

Fig. 3. Dataset extraction procedure. The labels at the bottom show the number of selected instances up to that step.



to what extent the Dockerfile is well-documented. Usually, developers tend to give an explanation comment of what each instruction does [3]. The last metric we introduced, *i.e., Usage of EXPOSE*, is boolean, and it checks the presence of the `EXPOSE` instruction. Such an instruction has the purpose of documenting the ports to be used when the Docker container will be executed[16].

Externally observable features. We defined two new metrics for the *Security/Best Practices* feature, *i.e., Image user is root* and *Num. of exposed secrets*. *Image user is root* is a binary metric that indicates whether the principal user of the image is root or not: A good security practice, indeed, is to use containers for which the main user does not have root privileges (*i.e.,* non-root user). *Num. of exposed secrets* measures the estimated number of secrets (*e.g.,* passwords or private keys) stored in the image: A good security practice is to avoid exposing sensitive data [30]. Therefore, the lower such a metric, the higher the security.

4 EXPLAINING DEVELOPERS' PREFERENCES

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

Hypotesis 1. Developers prefer images with fewer security issues.

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

Hypotesis 2. Developers prefer smaller images.

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

[16] https://docs.docker.com/engine/reference/builder/#expose

Hypotesis 3. Developers prefer official images.

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

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

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

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

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

A complex Docker image might result in low Dockerfile quality. Thus, we expect that a more complex Docker image leads to a higher number of security issues (among other issues), as it has been observed for normal source code [22]. Complexity metrics are related to the presence of security vulnerabilities, together with the developers' activity (*e.g.,* team size) [29]. Thus, we also expect that bad practices in the development process can increase security risks in the Docker images.

We do not formulate *hypoteses* regarding the officiality of the Docker image since the process behind the assignment of the "official image" badge is well-known[7].

5 EMPIRICAL STUDY DESIGN

The *goal* of the study is to understand which external features directly influence the developers' preferences and which configuration features indirectly do so (by directly influencing external features). The *context* consists in 2,441 open-source Docker images used as *base images* for 10 software applications hosted on GitHub, and on 299 Dockerfiles manually associated to a sample of Docker images from 2,441.

Our study is steered by the following research questions:

- RQ_1: Can the externally observable features explain the developers' preference for a Docker image? With this first research question, we want to know what external features, *i.e.,* those related to the Docker image, allow to explain the adoption and the preference expressed by the developers in terms of adoptions (how many times a Docker image is used as a base image in Dockerfiles) and perceived quality (prominence measured as the number of stars on DockerHub). This research question will allow to verify or disprove *hypoteses* 1, 2, and 3.
- RQ_2: Are configuration-related features correlated with externally observable features for Docker images? With the second research question, we want to understand which configuration features directly influence the external

features of a Docker image and, thus, indirectly influence the developers' preferences. This research question will allow to verify or disprove *hypoteses* 4 and 5.

5.1 Data Collection

The context of our study is composed of *objects*, *i.e.,* Docker images and their related Dockerfiles. In our study, we built two distinct datasets from the open-source codebase: D_{img} and D_{src}. D_{img} is composed of 2,441 instances of Docker images, associated with the respective number of adoption and the number of DockerHub stars. D_{src} contains a subset of the images from D_{img} (299) manually associated with the Dockerfiles used to build them. We use D_{img} for answering $RQ_{\mathbf{1}}$ and D_{src} for $RQ_{\mathbf{2}}$. The procedure we used for building such datasets is summarized in Fig. 3 and detailed below.

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

Mining Adoptions of Docker images. Our main objective with D_{img} is to annotate a set of Docker Images with their number of adoptions in downstream Dockerfiles and DockerHub stars. While the latter can be easily achieved by using DockerHub APIs, the former requires mining existing software repositories. To do this, we use *GHSearch* [6], which crawls data from open-spurce software projects hosted on GitHub providing metadata and statistics such as commits, contributors, stargazers and the other information related to the repository. We extracted the metadata for GitHub project repositories, as provided by the tool, starting from the date when Docker is introduced, *i.e.,* 2013, to January 2022. Next, we selected only the repositories where "Dockerfile" is among the language used to exclude projects that do not use Docker. As a result, we obtained a total of 50,487 projects. Then, we collected all the Dockerfiles from such projects (182,375, in total) and we extracted their content at the latest snapshot. We parse the Dockerfiles obtained, and we extract all the base images used (*i.e.,* the ones which follow the `FROM` instructions). As a result, we obtained a list of base images used. Finally, we get the unique images, and we count, for each of them, how many times they occurred. The final result is a set of 20,425 Docker images used as base images associated with the respective number of adoption (*i.e.,* how many times they appear in the `FROM` instructions).

Annotating Docker Images with Application. Besides having the number of adoption for the collected Docker Images, we also want to annotate them with the software they provide and its version. This is necessary because, to answer $RQ_{\mathbf{1}}$, we will need to group together all the images providing the same features and explain the developers' preferences among them, rather than among images providing different features. Indeed, let us imagine that we have two Docker images providing an Apache HTTP server, with 1,000 and 900 adoptions, respectively, and an image providing Nginx, with 2,000 adoptions. We do not know whether the higher number of adoptions is due to the fact that developers prefer the Docker image providing Nginx or they simply prefer Nginx. In other words, the number of adoptions between the two images providing Apache HTTP is comparable and might depend on the differences between the images, while the number of adoptions of the image providing Nginx can not be mixed with the others. The same is true for different versions of the same software: Developers might prefer a given version of Apache HTTP and base the choice on it rather than on the non-functional aspects of the Docker image. Therefore, we assigned each image with an *application name* (*e.g.,* "Tomcat") and an *application version* (*e.g.,* 7.0). To do this, we use a semi-automatic procedure. First, we removed all the instances where the Docker image repository name contains special characters that are not allowed by the Docker naming convention (*i.e.,* non-alphanumeric symbols or placeholders). Thus, from a total of 182,375 instances, we retain 141,583 of them. Next, we extracted the words contained in the image names by performing a string split over the separators (*i.e.,* dash or underscore). For example, from *alpine-maven-builder-jdk-8*, we extract the words *alpine*, *maven*, *builder*, *jdk* and *8*. The next step is to select, among all the obtained words, only those that are alphabetic (*i.e.,* do not contain symbols or numbers) and contain at least 3 characters. We do this to discard words that are not useful. Examples are *go*, *os*, *js* as we select Docker images containing applications and not OSs and programming languages. We selected all the words appearing in at least five image names, and we obtained a total of 338 unique words. Each of the selected words is a candidate application name. We discard word (*i.e.,* candidate applications) with less than five occurrences to avoid having too small groups for the analysis performed in $\mathrm{RQ}_{\mathbf{1}}$ and include software that is provided through a limited number of Docker images.

Next, we selected and assigned a set of tags (*i.e.,* clusters) to group each base image of our dataset by the contained application. For example, we assign the label *tomcat* to all the images that provide the tomcat web server. We used the dataset of Docker images obtained in the previous step to achieve this. At this point, a manual process is required to identify if a word corresponds to an application name to group similar Docker images (*i.e.,* clustering). This is done by manual annotation of all the extracted words that occur at least 5 times, *i.e.,* there are at least 5 unique Docker images containing those words, for a total of 338. Then, we manually check the candidate application names, and we select only the ones that are actual applications. We discard operating systems/Linux distributions (*e.g., ubuntu*, *debian*, *alpine*), programming languages (*e.g.,* python, java), and other commonly used words which do not pertain the application (*e.g., build*, *base*, *dev*, *runtime*, *aws*, *platform*). Examples of valid words we selected are *nginx*, *maven*, *jenkins*, *chrome*, *dotnet*, *envoy*, *mysql*. In some cases, different words could refer to the same application (*e.g., postgres* and *postgresql*). In such cases, we manually created clusters of names and associated them with a unique name (*e.g., postgres*, in the previous example). As a result, we obtain a total of 73 different applications associated with all names through which they appear in the Docker images. Finally, we associated each Docker image with a list of applications it provides by simply performing string matching with the words analyzed in the previous step. If a Docker image was associated with no application, we discarded it. This happened, for example, for Docker images providing Linux distributions, as previously explained. We manually analyzed cases in which a Docker image was associated with more than an application, and discarded the cases in which more than an application was actually provided. After this step, we obtain our final dataset of 2,776 Docker images (covering a total of 12,674 adoptions). We also annotate each image with the version of the application provided. To do this, we split the Docker image name as previously done to identify the application name, and we select the word with the highest number of numeric characters. We manually check if the version assigned to each image was correct.

Feature Extraction. We added to the dataset all the features needed to answer our research questions. Firstly, for each Docker image, we extracted the number of stargazers (*i.e.,* stars) by using the DockerHub APIs[17], to compute the perceived quality (*i.e.,* the prominence of a Docker image over the others, used as a dependent variable for $\mathrm{RQ}_{\mathbf{1}}$). We computed most of the metrics related to the external features from the literature we identified in Section 3.2, with some exceptions and small variations. We describe below only such cases. We did not consider the metric *Presence of temporary files smell*, because it can not be exactly measured automatically but only with a semi-automatic approach, as described in the reference article [23]. Moreover, we merged the metrics for the feature *Inherited Vulnerabilities* and *Packages Vulnerabilities* in *Num. of vulnerabilities*. We did this because, given a Docker image, we could not distinguish the layers inherited from the base images (*i.e.,* parent) and the additional layer added on top of them with the specific Dockerfile used, since we do not have such a Dockerfile in $\mathrm{D}_{\mathrm{img}}$. To compute the *Num. of vulnerabilities*, we used the *Clair* tool. For the metrics in the category *Security/Best Practices*, we adopt the *Whaler* tool, which returns *Image user is*

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

root and *Num. of exposed secrets*. To measure the *Officiality* feature, we implemented a web scraper to parse the presence of the label *"Official Image"* on DockerHub.

Table 3. Summary of the selected applications and sampled instances from the dataset.

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

5.1.2 Dataset of Dockerfiles associated with Docker Images (D_{src}). To perform the analysis required in the context of RQ_2, we need to have, for each Docker image, the source Dockerfile. Thus, we defined a second dataset, namely D_{src}, which contains a subset of the Docker images from D_{img}, in which each instance contains the content of the Dockerfile used to build it. To achieve this, we first randomly extracted a sample of D_{img} for the applications with the highest number of Docker images. We filtered D_{img} and selected only the Docker images for such selected applications obtaining a total of 299 instances. Manually annotating the Dockerfile from a Docker image is challenging: In most cases, a direct link to the Dockerfile is missing. Thus, we performed a random sampling selecting 299 total instances with a confidence level of 95% and 5% margin of error. Finally, we manually annotated the Dockerfiles related to each remaining Docker image. To achieve this, for each image, we looked at the DockerHub repository. If there was a direct reference to the Dockerfile, we assumed it was the one used to build it. Otherwise, we performed a Google search using the name of the image plus the word "Dockerfile" (*e.g., nginx Dockerfile*) looking for the source of the Dockerfile related to that image. If we obtained no results, we replaced the Docker image with another randomly selected, for the same application, to avoid hampering the representativeness of our sample. We report in Table 3 the total number of selected applications and the sampled number of instances, *i.e.,* the different groups of comparable Docker images and their number, involved in our experiment. In detail, we have 10 different groups having a number of Docker images varying from 12 (dotnet) to 344 (nginx). We have a total number of 2,441 open-source Docker images, and for a subset of them (299) we also have the source Dockerfile from open-source codebases..

Also in this case, we computed on D_{src} all the metrics related to the configuration features that were reported in Section 3.2, with some exceptions and small variations. We describe below only such cases. We excluded from the metrics related to the feature *Update Status* because we could not have a reliable measure for the metrics *Is base image up-to-date* and *Num. of out-of-date dependencies*. The update status of the base image and the package dependencies, indeed, depends on the time at which the adoption was made in the downstream Dockerfiles, and it changes over time. We cannot trace back the time at which one or more dependencies (possibly) became out-of-date in a Docker image and, thus, report if it was so at the time of adoption. Also, we do not compute the metric *Evolutionary trajectories category* [41]. This is because, in the original study, the authors show that this measure correlates with the build latency and the number of best practice violations, which we directly compute (*i.e., Build time*, *Num. of docker smells*, and *Num. of shell script smells*). For the metrics of the category *Script Quality*, we use the *Hadolint* tool to detect violations of best practices. For the other metrics, we use a modified version of the parser from the replication package of the analysis conducted by Schermann *et al.* [28]. Specifically, we added the extraction of code comments, as their parser does not retrieve them. For the metric of the *Project Activity* feature, we use the tool *PyDriller* to extract data from the source repository of each Dockerfile. For the *Build* category, we use the Python Docker wrapper[18] to build the Dockerfiles and measure their build time.

5.2 Experimental Procedure

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

5.2.1 RQ$_1$: Can the externally observable features explain the developers' preference for a Docker image? To answer RQ$_1$, we extract the external metrics described in our taxonomy (Fig. 2) on the dataset D$_{\text{img}}$. We removed all the instances with invalid metrics values (*e.g., Clair scanner* fails on some Docker images), obtaining a total of 2,441 valid instances for the analysis. Next, to evaluate what are the external features that affect the developer preferences for a Docker image, we build two mixed-effect generalized linear models [9]. In detail, we use the *lmer* function from the R library *lmerTest*. Each instance of the dataset contains the value of the metrics for the external features, the application name and version, the number of adoptions, and the number of DockerHub stars. We use as random effects the application name and version. We use as random effects the application name and version. In this way, different Docker images regarding the same application at the same version, are considered in the same group. We do this because we want to take into account the fact that developers might have different levels of preferences for Docker images that provide different software applications, based on the characteristics of the applications themselves, regardless of the other image-related factors evaluated in our study. For example, the images `jdk-8-alpine` and `jdk-8-slim` will be in the same group, while `jdk-9-slim` and `jre-8-slim` will belong to other groups. The dependent variables, or outcomes, are the following:

- **Number of adoptions**: the actual usage in software repositories of a Docker image (*i.e.,* objective preference), measured as the occurrences of a specific Docker image (*i.e.,* name and tag) in user-defined Dockerfiles (as reported before);
- **Number of DockerHub stars**: the number of stars of a Docker image reported on DockerHub. This measures the prominence of a Docker image over others expressed by the developers. The number of adoptions and the number of stars tend to be directly proportional ($r_s = 0.23$, *p-value* < 0.05).

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

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

[18] https://pypi.org/project/docker/

- *Is official*: the image is part of the Docker official images program, thus maintained following the official Docker guidelines.

All the independent variables refer to the external features described in Section 3.2. Before we performed the regression analysis, we applied some transformations to our dataset. First, we perform a correlation analysis to remove the highly correlated variables using a threshold of $r_s > 0.90$. None of the variables have been removed as their correlation coefficient remains below the threshold. Next, we computed the skewness coefficient of the distribution of all the variables. To normalize skewed distributions, we apply a logarithmic transformation to both dependent and independent variables (*i.e.,* $log(x + 1)$ since they are all non-negative. In our case, all the variable distributions are skewed (the lowest skewness value is 1.8, where a coefficient close to 0 means that the distribution is not skewed). Moreover, we apply a min-max normalization to fix the variables on the same scale. As a result of our analysis, for each variable of our model, we report the significance value (*i.e., p-value*), the standard error, the coefficients, and the polarity of the relationship of that coefficients. We consider a coefficient *important* for determining the developers' preferences if it is statistically significant, *i.e., p-value* < 0.05. To evaluate the model fit, we report the adjusted R^2, using the *rsq* R package. It describes the variation explained by the model. Moreover, we report the effect size, expressed by measuring the Pearson correlation coefficient between pairs of independent and dependent variables [7] for the cases in which the relation, reported by the model, is statistically significant (*i.e., p-value* < 0.05). We also report Cohen's d effect size magnitude, obtained from Pearson's r by using the formula $d = \frac{2*r}{\sqrt{1-r^2}}$ [27].

5.2.2 *RQ_2: Are configuration-related features correlated with externally observable features for Docker images?* To answer RQ_2, we compute the metrics related to the configuration features of our second dataset, *i.e.,* D_{src}. To perform the regression analysis on D_{src}, we built three mixed-effect generalized linear models. To explain how the external features are affected by the configuration features, we build a model for three of the external factors analyzed in RQ_1, as dependent variables, *i.e., Image size*, *Num. of vulnerabilities*, *Num. of exposed secrets*. We exclude from our regression modeling the external features *Is official* and *Image user is root* because the former is not an objective measure that depends on a set of non-quantifiable aspects, *i.e.,* is assigned by a team of Docker reviewers based on the official guidelines[7] and the latter can be directly controlled by the developer by adding a specific line of code. Also in this case, we consider the application name and version as a random effect. The independent variables (fixed effects in our models) are the metrics for the configuration features computed on the selected sample of Docker images (*i.e.,* D_{src}). In detail, the independent variables are the following:

- *Num. of docker smells*: number of best practice violations for Dockerfiles, extracted using the tool *hadolint*;
- *Num. of shell script smells*: number of best practice violations for shell script code used in Dockerfiles, extracted using the tool *hadolint*;
- *SLOC*: the total number of source lines of code (*i.e.,* without code comments and blank lines) in the Dockerfile;
- *Layer size*: the average number of commands executed in a single instruction block, to measure how much they are nested (*i.e.,* a proxy for the source code complexity);
- *Num. of docker instructions*: number of the used Docker instructions (*e.g.,* `RUN`, `FROM`, etc.) used in the Dockerfile;
- *Instructions entropy*: the Shannon entropy computed using the different Docker instructions used in the Dockerfile, as a measure for its complexity (*i.e.,* heterogeneity of the Dockerfile).;
- *Usage of additional script*: boolean flag that indicates whether or not the Dockerfile uses additional shell scripts, *i.e.,* it executes external scripts during the build of the Docker image;

- *Usage of external resources*: boolean flag that indicates whether or not the Dockerfile uses external resources, *i.e.,* it fetches additional data from remote sources (*i.e.,* URLs) during the build of the Docker image;
- *Usage of ENV*: boolean flag that indicates whether or not the Dockerfile uses environment variables, *i.e.,* identified by the instruction `ENV`;
- *Usage of build ARG*: boolean flag that indicates whether or not the Dockerfile uses build args, *i.e.,* identified by the instruction `ARG`;
- *Project age*: the age of the repositories that the Dockerfile belongs to, measured in seconds elapsed between the first and the last commit;
- *Num. of layers*: the number of layers that compose the Docker image, measured after the Dockerfile build;

Based on our hypotheses reported in Section 4, we define a model for each dependent variable, based on what we reasonably expect to impact each external feature. Specifically, for the outcome *Num. of exposed secrets*, we have as independent variables *Num. of docker smells*, *Num. of shell script smells*, *Instructions entropy*, *Usage of additional script*, *Usage of external resources*, *Usage of ENV*, and *Usage of build ARG*. For the outcome *Num. of vulnerabilities* we use as independent variables: *Num. of docker smells*, *Num. of shell script smells*, *SLOC*, *Usage of additional script*, *Usage of external resources*, *Usage of ENV*, *Project age*, and *Num. of layers*. Finally, for the outcome *Image size*, we have as independent variables: *SLOC*, *Num. of docker instructions*, *Layer size*, *Usage of additional script*, *Usage of external resources*, and *Num. of layers*.

We perform the same preprocessing steps done for answering RQ_1. First, we performed a correlation analysis to remove highly correlated variables (threshold of $r_s > 0.90$), but none were removed. Next, we evaluate the skewness coefficient. To normalize skewed distributions, we apply both square root and log transformations. In particular, we apply the log-transformation on the higher skewed distributions (skewness ≥ 1.8, *i.e.,* the metric *Num. of exposed secrets*), while the square root on the less skewed ones (skewness < 1.8). After this, we apply the min-max normalization to all of our variables. For each of our models, we compute the *p-value*, the standard error, the coefficients, and the polarity of the relationship of the coefficients with the dependent variable (*i.e.,* positive or negative). We consider a coefficient important for the dependent variable if the significance, *i.e., p-value*, is statistically significant (*p-value* < 0.05). As in the previous RQ, we compute the adjusted R^2 for each model, the effect size reported as Pearson's r between pairs of independent and dependent variables [7] and Cohen's d magnitude obtained from the correlation coefficient [27]. We do not report the results for all such models in the paper for readability reasons, but we discuss the main results, focusing on the relevant relationships we found. The detailed results are publicly available in our replication package [26].

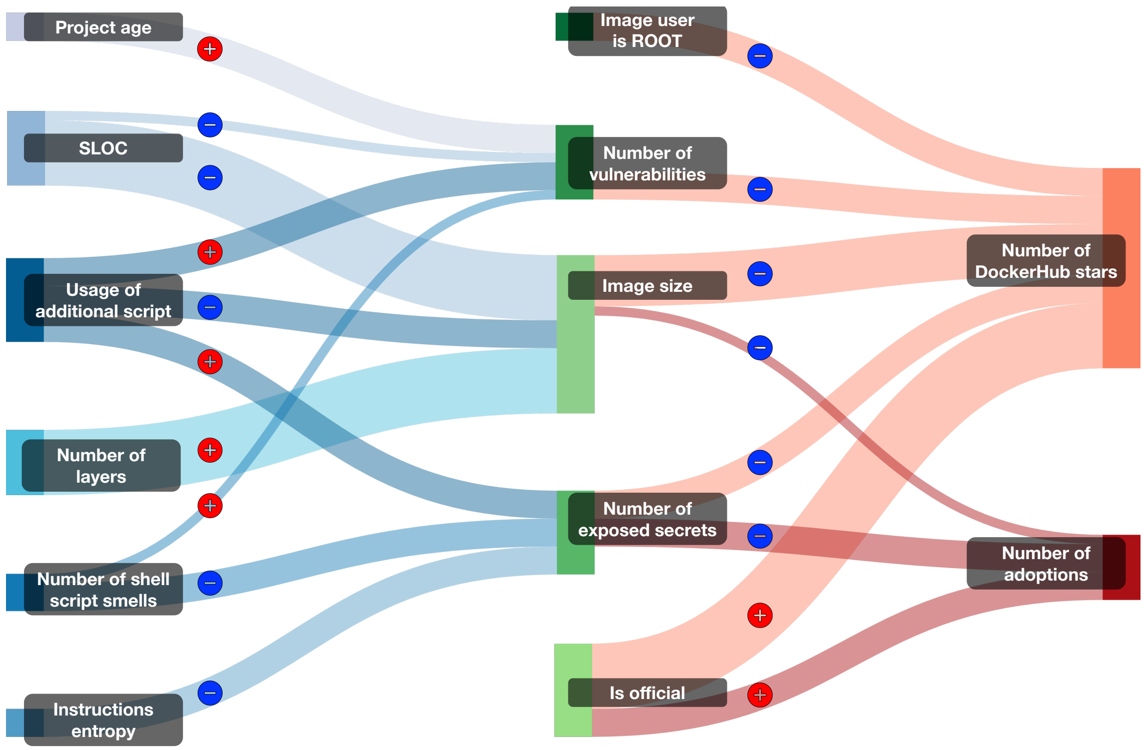
5.3 Replication Package

Both the datasets (D_{img} and D_{src}), along with the scripts we used to answer both our research questions, are publicly available in our replication package [26].

6 EMPIRICAL STUDY RESULTS

In this section, we report the results of our empirical study. Fig. 4 reports a summary of the relationships we found among configuration-related features and externally observable features, and then among external features and developers' preferences based on the results obtained from the two RQs. Connections indicate that the left-hand variable is significant in the model for explaining the right-hand variable. The size of the arrow represents the magnitude of the effect size

Fig. 4. Descriptive plot of the relation between configuration-related features, externally observable features, and preferences for Docker applications. The size of the arrow indicates the effect size magnitude (*i.e.,* very small, small, medium, or large). The polarity of the relationship is reported with plus (positive) and minus (negative) signs.



(*i.e.,* very small, small, medium, or high). The polarity of the relation is reported through a plus (positive) or minus (negative) sign.

6.1 RQ$_1$: Can the externally observable features explain the developers' preference for a Docker image?

We report in Table 4 the results of the performed regression modeling to explain the preferences for Docker images in terms of the number of adoptions and number of DockerHub stars, along with the Pearson's correlation between independent and dependent variables *Corr. Coeff*, and the effect size magnitude (*i.e.,* from Cohen's *d*). The variables *Num. of exposed secrets* and *Is official* are the most significant ones for the number of adoption, with a *p-value* < 0.001. That means developers tend to adopt official images, *i.e.,* images that follow the Docker official images program guidelines. This is also true when considering the number of DockerHub stars as a dependent variable. This can be a consequence of the fact that they have few exposed secrets with a lower number of vulnerabilities (Fig. 5). The metric *Image user is root* is not statistically significant for the outcome *Number of adoptions*. This means that it does not influence the usage of a Docker image. On the other hand, it is significant for the outcome *Number of DockerHub stars* with a negative relation. This means that image users prefer images where the main account is not *root*. Fig. 5 shows the relation

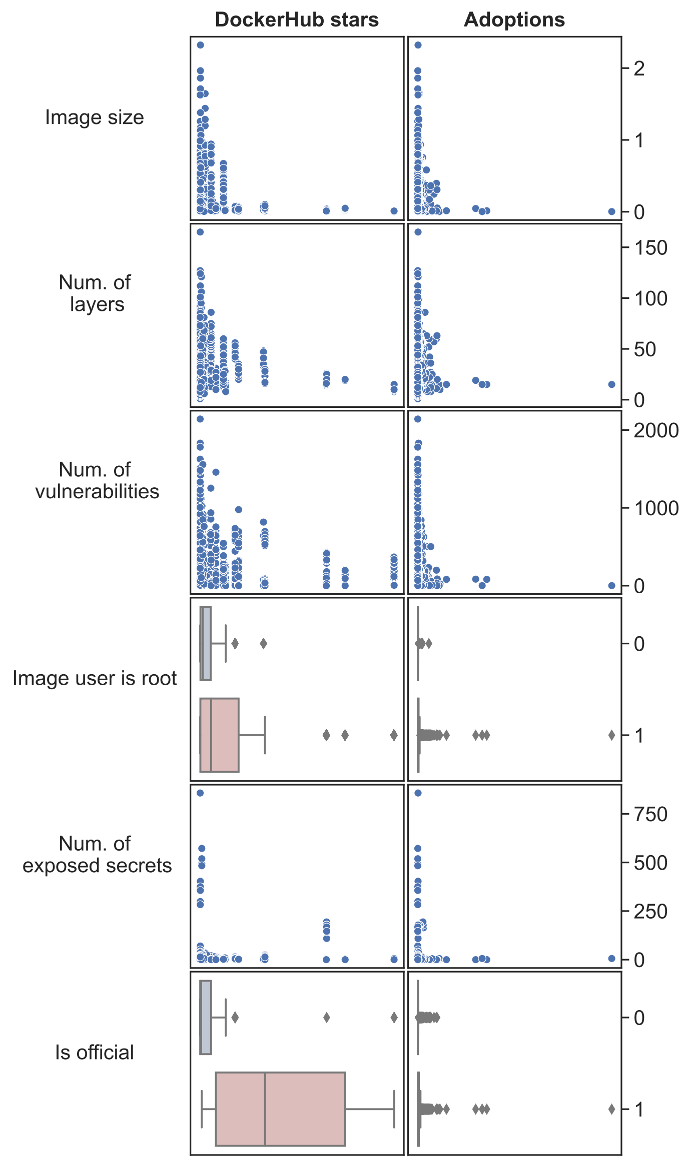


Fig. 5. Descriptive plot of the relationship between dependent and independent variables for the regression modeling of RQ_1.

between each independent and dependent variable involved in RQ_1. We use boxplots for binary variables and scatter plots for continuous ones. We have an overall inverse relation between independent variables and outcomes, the higher the adoptions, the lower the external features of the Docker images. We computed the Spearman correlation between dependent and independent variables. The number of stars has a negative correlation with *Image size* and a positive one with the metric *Is official*. This means that the developers prefer smaller images having the official image label. A heatmap with the correlation values can be found in our replication package [26].

Table 4. Mixed-effects models obtained for explaining developers' preferences through external factors. The columns *Corr. Coeff* and *Effect Size* report the value of Pearson's r and Cohen's d magnitude, respectively.

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

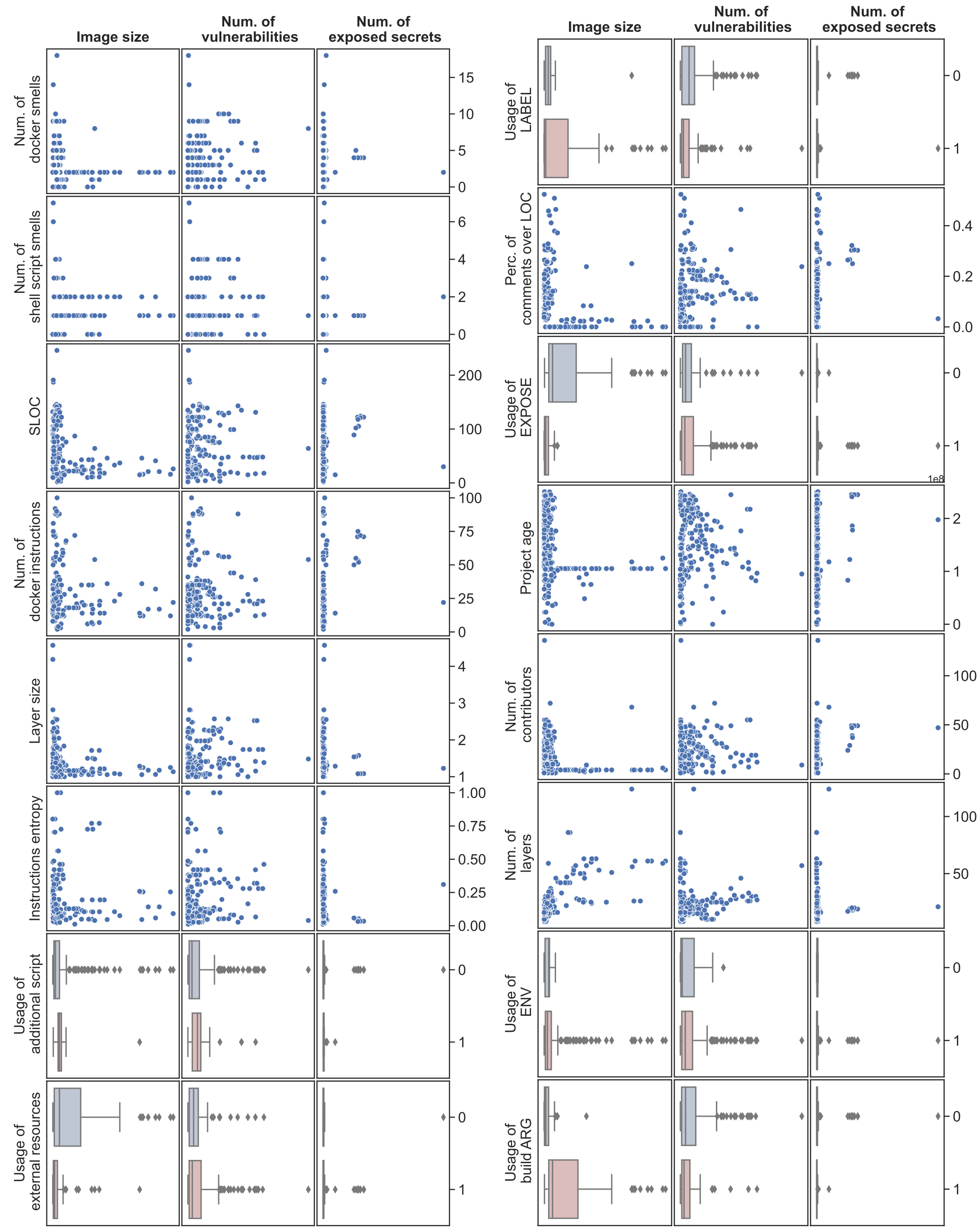
The adjusted R^2 for the two models are 0.18 (weak effect size) for the outcome *Number of adoptions*, and 0.74 (strong effect size) for the outcome *Number of DockerHub stars*. This shows that the external factors we considered are sufficient to explain the prominence of a Docker image over others expressed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

Summary of RQ₁. *Developers' perceived* (i.e., *prominence expressed in terms of DockerHub stars) and actual (in terms of adoptions) preferences can be explained by the image officiality-, security-, and size-related metrics. However, such metrics are much more effective in explaining the perceived preferences than the actual ones.*

6.2 RQ$_2$: Are configuration-related features correlated with externally observable features for Docker images?

We computed the Spearman correlation computed between configuration and external features of Docker applications. The highest correlation obtained is 0.75, between *Image size* with *Num. of layers*. When compared to *Layer size*, we have a negative correlation of −0.51. This means that large images have many layers that perform few actions, while in smaller images the number of layers is low and the number of actions performed is high. We also observe a negative correlation ($r_s = -0.28$) between *Usage of build ARG* and *Num. of exposed secrets*: This is reasonable since developers might use build arguments to pass secrets (*e.g.,* passwords or keys) instead of having them hard-coded in the Dockerfiles themselves. A heatmap with the correlation values can be found in our replication package [26].

When combining such metrics in the three models we investigated, first, we found that the number of exposed secrets in the Docker image (*Num. of exposed secrets*) is higher when the Dockerfile uses additional scripts (*Usage of additional script*) and has a lower number of shell smells (*Num. of shell script smells*). The latter can be counter-intuitive. This is because if there are additional scripts, external to the Dockerfile, it is likely that the shell-script code is in them instead of inside the Dockerfile. Moreover, it is unlikely that shell-script smells can expose secrets in Docker images. The adjusted R^2 for such a model is 0.40 (weak effect size). We also observed that vulnerabilities (*Num. of vulnerabilities*) occur more frequently in older projects (*Project age*), when Dockerfiles are bigger (*SLOC*), they use additional scripts (*Usage of additional script*), and they have more shell-script smells (*Num. of shell script smells*). The adjusted R^2 for such



(a) (b)

Fig. 6. Descriptive plot of the relation between dependent and independent variables for the regression modeling conducted in RQ$_2$.

a model is 0.24 (weak effect size). Finally, our results show that the image size highly depends on the number of layers (*Num. of layers*), as previously observed with the simple correlations. Similarly, the size is higher when the Dockerfile uses additional scripts (*Usage of additional script*) and fewer lines of code (*SLOC*). It is important to keep in mind that the size of a Docker image mainly depends on the number of layers and the base image used. For example, a Dockerfile that uses as base image the *nginx* web server, probably mainly performs the copy and the setup of the application to be contained. The adjusted R^2 is 0.76 (strong effect size). The detailed results of the models we built for RQ_2 are available in our replication package [26].

In summary, we observed that some configuration-related features have a significant role in explaining the external features we analyzed. In general, developers should keep the *SLOC* low to have benefits in terms of size and security. It is important to say that not all the lines of code (*i.e.,* instructions) have a direct impact on the image size (*e.g.,* the removal of non-functional instructions like `EXPOSE`). Similarly, developers should pay attention to the *Num. of layers*, which can negatively impact the size. Finally, the use of additional shell scripts should be discouraged since it has a negative impact on both the security (*Num. of exposed secrets* and *Num. of vulnerabilities*) and size (*Image size*). Also in this case there are exceptions, *i.e.,* not all the shell script smells directly lead to security issues.

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

6.3 Discussion

From the results of our study, we can extract several hints that benefit both researchers and developers interested in improving the quality of their Docker images. The general picture is described in Fig. 4, which summarizes the outcome of the regression modeling for both RQs. We observed that Docker images having the highest number of adoptions have a small storage size and a low number of layers. Also, the number of exposed secrets is low, along with a low number of shell script smells, also avoiding the usage of additional scripts. The number of SLOC has to be low, along with the heterogeneity of instructions (*i.e.,* entropy).

The officiality of the image is actually the strongest factor explaining the preference for Docker images, impacting both adoptions and stargazers count. For the latter, in addition to the features mentioned above, we have that image users prefer images with less number of vulnerabilities, where the main user of the image is not *root*. It is interesting to note that the number of vulnerabilities is positively affected by the repository age of the Dockerfile. This means, and confirms, that Dockerfiles must be actively maintained and updated to lower the presence of security vulnerabilities in the resulting images [30].

Also, the correlations found in our experiment are not strong for the specific metrics and features. Most likely, this happens because developers tend to pick official Docker images, with the assumption that they have the best quality overall [19]. We believe that this results from the fact that they are not aware of alternatives from the community of that images because it is difficult for users to compare similar Docker images as their peculiarities are not clearly highlighted [15]. An example is the `debezium/postgres:11` Docker image, where the source Dockerfile has fewer smells (*i.e.,* 6) compared to the official `postgres:11` (*i.e.,* 13). Another example is the `bitnami/nginx:1.19`, an unofficial Docker image for *nginx v1.12*, which has fewer security vulnerabilities (*i.e.,* 98) compared to the official image `nginx:1.19` (*i.e.,* 188). The behavior of the developers, when they pick a Docker image, could be related to the mismatch between

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

Fig. 7. Examples of Dockerfiles having different image sizes.

```
FROM openjdk:7-slim

# INSTALL REQUIREMENTS
RUN apt-get update
RUN apt-get install --no-install-recommends -y wget
RUN apt-get clean
RUN rm -rf /var/lib/apt/lists/*

# INSTALL TOMCAT
RUN wget http://archive.apache.org/dist/tomcat/tomcat-7/v7.0.69/bin/
    ↪ apache-tomcat-7.0.69.tar.gz -O tomcat.tar.gz
RUN tar zxf tomcat.tar.gz
RUN rm tomcat.tar.gz
RUN mv apache-tomcat* tomcat

# ADD TOMCAT EXECUTABLE TO PATH
ENV PATH "$PATH:/tomcat/bin"

EXPOSE 8080

CMD ["catalina.sh", "run"]
```

(a)

```
FROM openjdk:7-slim

# INSTALL REQUIREMENTS
RUN apt-get update && \
    apt-get install --no-install-recommends -y wget && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

# INSTALL TOMCAT
RUN wget http://archive.apache.org/dist/tomcat/tomcat-7/v7.0.69/bin/
        ↪ apache-tomcat-7.0.69.tar.gz -O tomcat.tar.gz && \
    tar zxf tomcat.tar.gz && \
    rm tomcat.tar.gz && \
    mv apache-tomcat* tomcat

# ADD TOMCAT EXECUTABLE TO PATH
ENV PATH "$PATH:/tomcat/bin"

EXPOSE 8080

CMD ["catalina.sh", "run"]
```

(b)

```
FROM openjdk:7-slim

# INSTALL TOMCAT
RUN apt-get update && \
    apt-get install --no-install-recommends -y wget tomcat7 && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

# ADD TOMCAT EXECUTABLE TO PATH
ENV PATH "$PATH:/usr/share/tomcat7/bin"

EXPOSE 8080

CMD ["catalina.sh", "run"]
```

(c)

adoptions and image preferences (*i.e.,* prominence), where we have a Pearson correlation $r = 0.25$ and medium effect size. We believe that, for the same reason, official Docker images tend to have more stars, *i.e.,* higher prominence ($r = 0.66$ and large effect size).

We can summarize some takeouts from the results of our empirical study.

***Image size* is influenced by *Num. of layers*.** Considering the results of our analysis, the number of LOC influences the number of layers. In Fig. 7 we report three different examples to qualitatively assess this relation. We have the Dockerfile *a* and a version with the number of layers reduced (*i.e.,* Dockerfile *b*) maintaining the same number of lines. Thus, if we build Dockerfile *a*, the resulting image will have 21 layers with a size of ~315 MB. If we build Dockerfile *b*, the resulting image will have 15 layers with a size of ~282 MB.

In some cases, if a Dockerfile downloads an external package, the size of the resulting image will change independently of the number of layers and lines of code. For example, if we consider Dockerfile *b* with the two `RUN` instructions merged, compared to Dockerfile *c* where *tomcat* is installed using via *apt-get*, the resulting images will have the same number of layers, but the size of the former is higher than the latter (282 MB vs. 277 MB). Moreover, looking at Dockerfile *a* and *b*, it is clear that the number of layers is not related to the number of LOC but to the number of Docker instructions However, we show an example where we modify instructions that directly impact the composition of the final image. The same does not apply to some kind of instructions, *i.e.,* removing instructions such as `LABEL` or `EXPOSE`. To the best of our knowledge, there are no automated tools for the refactoring of Dockerfiles that can help to reduce the image size. However, there is the *docker-slim* tool[20] that does not act on the Dockerfile, but directly on the container. It creates a slimmed-down version of the Docker container maintaining the same functionalities.

Shell scripts can be a proxy for security issues. An interesting point to discuss, resulting from our empirical study, is the fact that the usage of shell scripts can lead to security issues. There are mainly two types of shell scripts used in Dockerfiles: Embedded shell scripts and external shell scripts. For the former, the major issues are related to the best practice violations detected with the *hadolint*. For the latter, the main issue is that the shell script is executed in the same build context as the Docker image. In this way, it is possible to inject malicious code or access the host file system. In general, shell script code must be written in a safe way, following best practices, and additional scripts must be checked, or else they must come from trusted sources. It will be better to avoid copy-paste shell scripts from random websites. An example of a best practice violation that can expose the Docker container to security issues is the rule violation identified as *SC1098*, detected by the tool *hadolint*. The violation concerns the missing quote/escape for special characters when using the `eval` command. This rule is not a security issue itself, but its violation can lead to unpredictable outcomes from the script code. This can be exploited to inject malicious code[21] Moreover, the main proxy for security vulnerabilities is related to the update status of the Docker images, *i.e.,* most updated images usually have fewer security vulnerabilities, but are not exempt from them [30, 39].

Dockerfile smells do not explain the adoption of the final Docker image. In the current scientific literature, the main measure to evaluate the quality of Docker images [5, 34] is the number of best practice violations (*i.e.,* smells) detected by the *hadolint* tool. Our results show that Dockerfile smells are not relevant for explaining any of the external factors we considered. In other words, their impact on the developers' preferences, when they have to choose whether they should adopt a Docker image, is negligible. It is possible that the current catalog of smells is still not sufficiently complete, or else only some of them are relevant for explaining the adoption of Docker images. Future work should be aimed at finding new types of smells, more related to the impact that they have on the resulting Docker image.

[20] https://github.com/docker-slim/docker-slim

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

7 THREATS TO VALIDITY

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

Construct Validity. We use state-of-the-practice tools such as *Clair* and *Hadolint*, to compute some of the metrics related to both external and configuration features (*e.g., Num. of vulnerabilities* and *Num. of docker smells*). To the best of our knowledge, the effectiveness of such tools for detecting the aspect that they aim at capturing was not validated in any previous study. However, such tools are already adopted both by developers in practice and researchers [5, 30, 34].

Internal Validity. To build our datasets we relied on the tool *GHSearch*, which provides all the software repositories from GitHub having more than 10 stars. While this could have biased the results towards more popular projects, we used this procedure to minimize the number of toy projects (*e.g.,* students' tests with Docker) in our datasets. While assigning the application name and version to each Docker image, we excluded the ones that contained more than an application name. We did this to avoid Docker images providing unique features that no other images could provide (*i.e.,* not comparable in terms of the environment alone). In doing so, we discarded 205 Docker images, which is negligible. An example of a discarded image is `tiangolo/uwsgi-nginx-flask:python3.5`[22]. It is worth saying that we only selected Docker images containing applications, so we discarded images for programming languages and OSs. Thus, we excluded a total of 128,704. Moreover, tagging some of the common programming languages and operating systems following the same procedure of Section 5.1, among the excluded images, we have 56,792 and 42,296, respectively. The remaining are uncategorized. In the first study, we ran a literature review to extract a collection of quality metrics that can impact the perceived quality of Docker images. We did not perform a Systematic Literature Review (SLR) on Docker quality to build the taxonomy because the topic is too broad and it would have been outside the scope of this paper. This is why we have not followed all the guidelines typically used to run a SLR [17]. As a result, we could have unintentionally excluded from our study some metrics defined in the literature relevant four our study. However, we still tried to minimize this by using some of the guidelines defined by Kitchenham and Brereton [17]: First, we use precise inclusion and exclusion criteria (Table 1) to make sure we do not select irrelevant papers. Second, to enlarge the initial set of papers we selected, we both used snowballing (to include older relevant literature) and searched for papers that cite them (to include more recent literature).

External validity. Because of the procedure we used to build D_{img}, we started from Dockerfiles of downstream applications to define a list of Docker images to analyze. It is possible that, because of this process, we ignored some Docker images that are not used in open-source software but are used in proprietary software, such as *Oracle db*[23]. While it is clear that we could not have captured the number of adoptions for them without having access to a large amount of proprietary Dockerfiles, it is true that we could have done so for the number of stars, which is always publicly available. We decided not to have two different datasets for the two dependent variables used to answer RQ_1 to avoid obtaining incomparable results. For D_{src}, we manually looked for the Dockerfiles of a sample of Docker images provided for the top ten applications in terms of the absolute number of Docker images available. The results of RQ_2 might not generalize to all the applications we consider. Still, this procedure allowed us to cover about ~50% of the total number of Docker image usages. It is important to clarify that our study was conducted on open-source Docker images and Dockerfiles, and, thus, our findings should not be generalized to other contexts (*e.g.,* industrial projects). In addition, the results come from a correlational study, where we cannot infer causality based on the data alone. In general, we reported practical examples to support our findings.

---

[22] https://hub.docker.com/r/tiangolo/uwsgi-nginx-flask
[23] https://hub.docker.com/_/oracle-database-enterprise-edition

8 CONCLUSION AND FUTURE WORK

Containerization is widely adopted in practice, and Docker is the leading technology. There are plenty of Docker images available in public repositories such as DockerHub, some of which provide the same software systems. It is unclear what aspects influence developers' preferences. In this paper, we first performed a literature review of 31 papers to find what are the *externally observable features* and *configuration-related features* factors typically considered. As a result, we defined a taxonomy of such features, along with the metrics typically used to measure them. Next, using such metrics, we performed an empirical study on a dataset of 2,441 Docker images to evaluate (i) what externally observable features impact the adoption of Docker images, and (ii) to what extent the configuration features influence external features. Our results show that the developers prefer Docker images that are official, secure, and small in storage size. Moreover, in terms of configuration features that are a significant impact on them, the *Num. of layers* must be kept low and *Usage of additional script* must be avoided if possible, where also the number of *Num. of shell script smells* must be low. Based on these results, future research could be aimed at defining a quantitative score for measuring the quality level of Docker images and Dockerfiles. Such a score could allow (i) developers to choose among different alternative Docker images, and (ii) researchers to build automated tools that take quality into account by objectively measuring it.

9 ACKNOWLEDGEMENTS

The authors would like to thank Marco Russodivito and Stefano Fagnano for their precious help during the data extraction process.

REFERENCES

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

[2] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is more: quantifying the security benefits of debloating web applications. In *28th USENIX Security Symposium (USENIX Security 19)*. 1697–1714.

[3] Hideaki Azuma, Shinsuke Matsumoto, Yasutaka Kamei, and Shinji Kusumoto. 2022. An empirical study on self-admitted technical debt in Dockerfiles. *Empirical Software Engineering* 27, 2 (2022), 1–26.

[4] Antonio Brogi, Davide Neri, and Jacopo Soldani. 2017. DockerFinder: multi-attribute search of Docker images. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 273–278.

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

[6] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling projects in github for MSR studies. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 560–564.

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

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

[9] Andrew Gelman and Jennifer Hill. 2006. *Data analysis using regression and multilevel/hierarchical models*. Cambridge university press.

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

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

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

[13] 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, 1148–1160.

[14] Zhuo Huang, Song Wu, Song Jiang, and Hai Jin. 2019. Fastbuild: Accelerating docker image building for efficient development and deployment of container. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 28–37.

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

[16] Shinya Kitajima and Atsuji Sekiguchi. 2020. Latest Image Recommendation Method for Automatic Base Image Update in Dockerfile. In *International Conference on Service-Oriented Computing*. Springer, 547–562.

[17] Barbara Kitchenham and Pearl Brereton. 2013. A systematic review of systematic review process research in software engineering. *Information and software technology* 55, 12 (2013), 2049–2075.

[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, 781–791.

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

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

[21] 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 *European Symposium on Research in Computer Security*. Springer, 257–276.

[22] Francesco Lomio, Emanuele Iannone, Andrea De Lucia, Fabio Palomba, and Valentina Lenarduzzi. 2022. Just-in-time software vulnerability detection: Are we there yet? *Journal of Systems and Software* (2022), 111283.

[23] Zhigang Lu, Jiwei Xu, Yuewen Wu, Tao Wang, and Tao Huang. 2019. An empirical case study on the temporary file smell in dockerfiles. *IEEE Access* 7 (2019), 63650–63659.

[24] Antony Martin, Simone Raponi, Théo Combe, and Roberto Di Pietro. 2018. Docker ecosystem–vulnerability analysis. *Computer Communications* 122 (2018), 30–43.

[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*. 476–486.

[26] Giovanni Rosa, Simone Scalabrino, Gabriele Bavota, and Rocco Oliveto. 2020. *Replication package*. https://doi.org/10.6084/m9.figshare.20131727.

[27] John Ruscio. 2008. A probability-based measure of effect size: robustness to base rates and other factors. *Psychological methods* 13, 1 (2008), 19.

[28] Gerald Schermann, Sali Zumberi, and Jürgen Cito. 2018. Structured information on state and evolution of dockerfiles on github. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 26–29.

[29] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason Osborne. 2011. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Trans. Software Eng.* 37 (11 2011), 772–787. https://doi.org/10.1109/TSE.2010.81

[30] Rui Shu, Xiaohui Gu, and William Enck. 2017. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. 269–280.

[31] Dimitris Skourtis, Lukas Rupprecht, Vasily Tarasov, and Nimrod Megiddo. 2019. Carving perfect layers out of docker images. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*.

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

[33] Byungchul Tak, Hyekyung Kim, Sahil Suneja, Canturk Isci, and Prabhakar Kudva. 2018. Security analysis of container images using cloud analytics framework. In *International Conference on Web Services*. Springer, 116–133.

[34] 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.

[35] Yiwen Wu, Yang Zhang, Tao Wang, and Huaimin Wang. 2020. An empirical study of build failures in the docker context. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 76–80.

[36] 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, 185–190.

[37] Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles, and Jesus M Gonzalez-Barahona. 2019. On the impact of outdated and vulnerable javascript packages in docker images. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 619–623.

[38] Ahmed Zerouali, Tom Mens, and Coen De Roover. 2021. On the usage of JavaScript, Python and Ruby packages in Docker Hub images. *Science of Computer Programming* 207 (2021), 102653.

[39] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus M Gonzalez-Barahona. 2019. On the relation between outdated docker containers, severity vulnerabilities, and bugs. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 491–501.

[40] Yang Zhang, Huaimin Wang, and Vladimir Filkov. 2019. A clustering-based approach for mining dockerfile evolutionary trajectories. *Science China Information Sciences* 62, 1 (2019), 1–3.

[41] Yang Zhang, Gang Yin, Tao Wang, Yue Yu, and Huaimin Wang. 2018. An insight into the impact of dockerfile evolutionary trajectories on quality and latency. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 138–143.

[42] 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.

[43] 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, 1–10.