An Empirical Investigation Into the Use of Dockerfile Preprocessors for Docker Image Management

Wataru Mabuchi
NAIST, Japan
mabuchi.wataru.mx8@is.naist.jp

Yutaro Kashiwa
NAIST, Japan
yutaro.kashiwa@is.naist.jp

Kenji Fujiwara
Nara Women's University, Japan
kenjif@ics.nara-wu.ac.jp

Hajimu Iida
NAIST, Japan
iida@itc.naist.jp

***Abstract*—Docker plays a crucial role in providing uniform software development. Many Docker development projects deliver multiple images in order to support various users who need different base images, versions, and architectures. To do so, the projects need to develop different contents of Dockerfiles for each support. For example, if developers provide their product on different Linux OSs, Dockerfiles need to contain package installing commands with an appropriate package manager for each Linux OS. To reduce the development tasks, many projects often develop their own tool to generate multiple Docker images automatically (hereafter, Dockerfile Preprocessors). However, it is still not clear how the projects adopt Dockerfile Preprocessors and what the benefits are.**

This study explores the characteristics of projects using Dockerfile Preprocessors, the timing, impact, and purpose, and the maintenance effort of using Dockerfile Preprocessors. Our empirical results show that (i) there is "Container build" pattern that does not generate multiple Dockerfiles; (ii) Projects using DPPs have more tags, supported Docker images, and architecture supports than projects without DPPs; (iii) 66% of projects develop DPPs in the middle of development; (iv) the common reasons for adopting DPP is to reduce the effort of creating Dockerfiles, and to ease updating versions/variations/architectures; (v) the adoption of DPPs does not increase releasing activities.

***Index Terms*—Docker images, Dockerfiles, Preprocessors**

I. Introduction

Docker is the de-facto standard platform for container virtualization [1], being widely adopted in open-source software development [2] and commercial software development [3]. In particular, Docker is often used for software distribution because of its high resource efficiency and portability [4] [5]. Docker Hub, which is the official community for container distribution [6], hosts more than 1,700,000 container images [7], constituting large ecosystems. The container development community is active and still expanding, and so daily 1,241 new images are pushed on average [8].

Many Docker development projects support multiple environments. For example, the Python project provides different base images (*e.g.,* Ubuntu or CentOS), versions (*e.g.,* 3.7 or 3.11), and architectures for the host machine (*e.g.,* AMD64 or i386). These factors are independent and so a Docker image consists of a combination of different environmental factors. Depending on the combination, the code to be written in Dockerfiles varies slightly or sometimes significantly. For instance, Dockefiles using Alpine and Debian as base images need to use `apk` and `apt-get`, respectively.

Developers often create self-made scripts (hereafter called Dockerfile Preprocessor) to maintain multiple Dockerfiles because it is tedious, time-consuming, and even prone to human error to write Dockerfiles for all the combinations. Dockerfile Preprocessor (DPP) automatically generates different Dockerfiles for each combination. DPP plays an important role in Docker image release management in practice but, to the best of our knowledge, only a few studies have analyzed them. Oumaziz *et al.* [9] performed an empirical study on code clones of Dockerfiles. They observed DPPs in 68.5% of repositories and classified them into three patterns. While their studies revealed patterns of DPPs, it is unclear how DPPs are employed and contribute to the development.

This study investigates the use of DPPs; specifically, studying the characteristics of projects using DPPs, the timing, impact, purpose, and maintenance effort of using DPPs. We manually inspected the 146 Docker official projects[1] and performed a qualitative and quantitative analysis. This paper makes the following contributions:

1) We conduct a qualitative analysis and find a unique type of DPPs that do not generate multiple Dockerfiles (*i.e.,* "Container build" type), which is not observed in the previous study. Also, throughout our manual inspections, we observe various reasons for adopting DPPs; (i) to automate creating Dockerfiles (ii) to support multiple variants, (iii) versions, and (iv) host CPU architectures.
2) We report a quantitative analysis discussing the characteristics of projects using DPPs, the timing of adopting DPPs, and the impact, as well as the maintenance cost of adopting DPPs. We find that (i) projects using DPPs have more tags, supported Docker images, and architecture supports than projects without DPPs; (ii) most DPPs are developed in the middle of development; (iii) adopting DPPs does not impact their release activities (iv) DPPs require the maintenance effort, specifically, projects make 22 changes in DPPs per 10 changes in Dockerfiles.
3) We make publicly available the labeling results of our manual inspection and the used scripts.

[1]https://docs.docker.com/trusted-content/official-images/

Replication package. To facilitate replication and further studies, the data used in this study are publicly available.[2]

Paper Organization. Section II introduces the background of this study. Section III outlines the data collection process. Section IV describes our research questions encompassing the motivation, approach, and results. Section V discusses the summary of findings and implications. Section VI introduces related work and Section VII discusses the threats to validity. Finally, Section VI summarizes our findings.

II. Background

This section introduces the background of Docker, Dockerfile, Dockerfile Preprocessor, and our motivating example.

A. Docker and Dockerfile

Docker is an open platform for building, running, managing, and shipping containerized applications. Docker containers can be isolated from the host infrastructures, which enables developers to ship their products easily and quickly [10] [11] [12]. In addition, Docker has another advantage of reproducing containers easily and steadily, based on Dockerfile [13]. Dockerfile is a text document that follows a specific format and contains a set of instructions. For example, the `FROM` instruction creates a new build stage imported from a base image that is already built. Another example is the `RUN` instruction that can perform any commands contained in the current stage [14]. Listing 1 shows an example of Dockerfile to build Python 3.11. In the figure, line 5 installed a build tool (*i.e.,* `.build-deps`) with `apk` and line 7 downloads a compressed file using `wget`. By building these downloaded files in the `RUN` instruction, Python 3.11 can be reproduced on Docker images.[1]

In recent years, most of the projects have supported multiple environments and published various images that support different base images, versions (of base images and the product itself), and architectures. Even if developers implement Dockerfile for the same product (*e.g.,* Python) but use base images, the contents of Dockerfile differ slightly or sometimes significantly. Listing 2 shows an example of Dockerfile to build Python 3.7. By contrast to Listing 1, Listing 2 assigns a different value to the environmental variable (*i.e.,* `PYTHON_VERSION`) in line 3 (*i.e.,* 3.11.1 and 3.7.16). More importantly, looking at line 1 in Listing 1 and Listing 2, Alpine and Buster are specified in the Dockerfile, respectively. Because of this difference in the base images, the following commands need to be replaced. Alpine and Buster are different Linux distributions and use different package managers. For example, in both figures line 5 has different commands to install packages but Alpine is required to use `apk` while Buster is required to use `apt-get`.

For every release, in many projects, developers manually modify their Dockerfiles, which results in defects. Figure 1 shows an example of defects in Dockerfiles. The project had to update environmental variables for the version of the product

Listing 1: Dockerfile for Python 3.11 based on Alpine

```
1 FROM alpine
2 ..
3 ENV PYTHON_VERSION 3.11.1
4 RUN ..
5    apk add --no-cache --virtual .
      build-deps
6    ..
7    wget python.tar.xz "https://../
      $PYTHON_VERSION.tar.xz";
8 ..
9 CMD ["python3"]
```

https://github.com/docker-library/python/blob/6aba522/3.11/alpine3.17/Dockerfile

Listing 2: Dockerfile for Python 3.7 based on Buster

```
1 FROM buster
2 ..
3 ENV PYTHON_VERSION 3.7.16
4 RUN ..
5    apt-get update;
6    apt-get install -y
      --no-install-recommends
7    patchelf;
8    wget python.tar.xz "https://../
      $PYTHON_VERSION.tar.xz";
9 ..
10 CMD ["python3"]
```

https://github.com/docker-library/python/blob/6aba522/3.7/buster/Dockerfile

written in Dockerfile but this was not done. The Dockerfile seems to be copied and pasted from 2.4.0 in a commit.[3] After that another developer reported several variables had not been updated. Developers have to update many variables in not only Dockerfiles but also configure files etc. Therefore, there is a high risk of inducing bugs.

B. Dockerfile Preprocessor (DPP)

To reduce the risks of human errors and to minimize the effort of updating many variables and files, in recent years, many projects have developed their own tools (hereinafter called, Dockerfile Preprocessor) to automate updates of Dockerfiles. Dockerfile Preprocessor (DPP) is a script set that automatically generates Dockerfiles for multiple environments. DPPs take each set of versions, base images, architectures, etc. as input and simultaneously generate multiple Dockerfiles that contain those combinations of inputs placed in appropriate locations.

In a previous study, there are three types of DPPs [9]: Template processor, Find and replace, and Generator. These three types of DPPs are introduced as follows.

Template processor pattern: The "Template processor" pattern is a method of generating a Dockerfile replacing variables in the Dockerfile template according to variable information. It has two types of files: templates and generation

[2]https://github.com/sdlab-naist/scam2024_replication_package

[3]https://github.com/influxdata/influxdata-docker/commit/8b784b47

[4]https://github.com/influxdata/influxdata-docker/issues/651

Fig. 1: An example of human error[4]

scripts. Generation scripts replace variables in the templates from the generation scripts with commands such as "sed". As an example of template substitution, NGINX container management project is shown in Listing 3. The Dockerfile is generated by replacing variables in the template (such as `%%ALPINE_VERSION%%`) with variable information in the script (such as `$alpinever`) using the "sed" instruction.

Find and replace pattern: The "Find and replace" pattern generates a Dockerfile using a replacement command such as "sed" as in the Template processor method. However, the files used as templates are different. Template processor uses a prepared template, while find and replace uses a previously created Dockerfile.

Generator pattern: The "Generator" pattern generates a Dockerfile from empty files dividing Dockerfile into parts. The full text of the Dockerfile is divided by descriptions that change with variable information, such as FROM statements and package manager-related commands, and each generation method is declared. Then, the Dockerfile is generated by using different methods according to the variable information and processing variables within the methods. Examples of the generation scripts in the `ibmjava` project are shown in Listings 4 and 5. Listing 4 is the method declaration part. In the method, the package manager's instructions, which vary depending on the base image, are written to the Dockerfile with "cat" instruction. Listing 5 is the method invocation part. The method to be executed is changed according to the base image. As described above, a Dockerfile is generated by writing statements corresponding to variable information to a blank file using the "cat" instruction.

C. Motivating Example

Oumaziz *et al.* [9] classified the types of DPPs, but still it is not clear how developers decided to develop DPPs and how DPPs contribute to DPPs. In fact, some questions on Stack Overflow ask about the effectiveness of DPP adoption.[5] In the post, a developer asked about the reasons and advantages of using one of the files constituting DPP ("Dockerfile.template"), but no clear answer has been made (as of June 2024).

[5]https://stackoverflow.com/questions/77316398/dockerfile-template-vs-dockerfile

Listing 3: Example of Template processor pattern

```
sed -i \
  -e 's,%%ALPINE_VERSION%%,'"$alpinever"',' \
  -e 's,%%DEBIAN_VERSION%%,'"$debianver"',' \
  ..
```

https://github.com/nginxinc/docker-nginx/blob/d039609e/update.sh

Listing 4: Example of Generator pattern (declaration part)

```
print_ubuntu_java_install() {
  ..
  cat >> $1 <<'EOI'
RUN set -eux; \
    ARCH="$(dpkg --print-architecture)"; \
    case "${ARCH}" in \
EOI
  print_java_install ${file} ${srcpkg} ${
    dstpkg};
}
```

https://github.com/ibmruntimes/ci.docker/blob/53aa230/ibmjava/update.sh

Listing 5: Example of Generator pattern (invocation part)

```
if [ "${os}" == "ubuntu" ]; then
      print_ubuntu_java_install ${file} $
        {srcpkg} ${dstpkg};
elif [ "${os}" == "alpine" ]; then
      print_alpine_java_install ${file} $
        {srcpkg} ${dstpkg};
elif..
```

https://github.com/ibmruntimes/ci.docker/blob/53aa230/ibmjava/update.sh

As there are no on-the-shelf DPP tools [9], developers need to develop by themselves. Also, the DPPs might require the maintenance costs. Revealing the benefits and costs of adopting DPPs would provide developers with materials to decide whether they should develop DPPs. To do so, this study aims to reveal the use, benefits, and costs of DPPs in practice.

III. Data Collection

Figure 2 shows an overview of the data collection process. Our data collection utilizes the repositories of Docker official images on GitHub [15]. The official Docker organization selects high-quality images to be entered into the repository. All relevant information is gathered in this repository. Specifically, this repository contains the library definition files for the images. The library definition file contains the repository URL for each image and the meta information about their provided images. We first analyze each file, obtain the repository URL, and clone each repository, for all the library definition files.

Next, we identify the projects using DPPs. To do so, we manually inspect each of the cloned repositories. Specifically, two of the authors manually and independently inspect all the files contained in each repository. Furthermore, if the

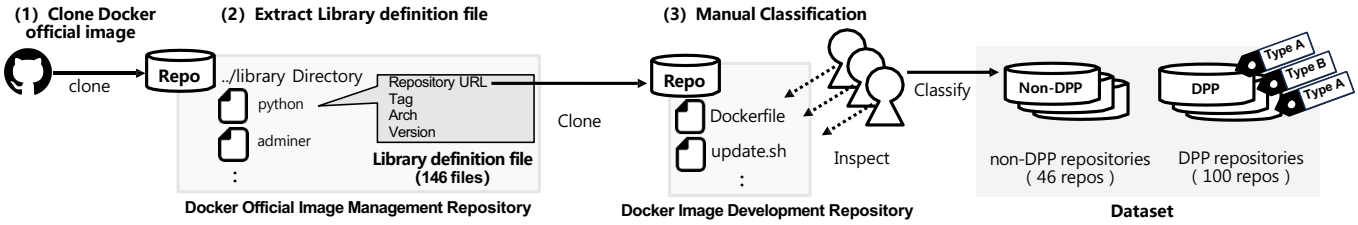


Fig. 2: Overview of the data collection process

repository has a DPP, we categorize it into three types of DPPs defined by Oumaziz *et al.* [9] and new types.

For easier categorizations, we decided to make and follow the following rule. We first check if the repository has a Dockerfile template. If the template exists in the repository, we regard the DPP as a "Template processor" pattern. Then we find scripts that update Dockerfile. If it exists, we categorize the DPP as a "Find and replace" pattern. If the scripts generate a Dockerfile from scratch, we classify the DPP into a "Generator" pattern. If the inspected DPPs cannot be categorized into any types, we assign them with a new label.

When two inspectors assigned a different label to the repository (*i.e.,* None, Template, Find and Replace, Generator, and new types), we asked another inspector (a collaborative researcher who is familiar with Docker studies) to solve the conflicts. The three annotators have 2 to 9 years of Docker experience (2, 7, and 9 years, respectively).

It is worth noting that we decided not to use the DPP detection approach of the previous study [9]. The previous study explored projects that have duplicate Dockerfiles. However, not all the projects make duplicate Dockerfiles. To prevent missing new types of DPPs, we inspected all the repositories of Docker official images, even though this required significant time and effort.

IV. Research Questions

This study aims to understand the benefit of using DPPs and address five research questions, investigating the characteristics of projects using DPPs (RQ1), the timing (RQ2), impact (RQ3), motivation of adopting DPPs (RQ4), and effort of maintaining DPPs (RQ5). The following sections first report the results of manual inspection (RQ0), and then describe the motivation, approach, and the obtained results for each research question.

A. RQ_0: Are there different types of DPPs than the previous study by Oumaziz et al.?

Motivation. Although Oumaziz *et al.* [9] categorized DPPs into three types, the studied projects are only those that have multiple Dockerfiles. Therefore, if there are DPPs that do not generate multiple Dockerfiles, the previous study might have missed other types of DPPs. Throughout our manual inspection, we would like to investigate whether other types of DPPs exist.

Results. We manually inspected all the repositories included in Docker official images and found 100 repositories that adopt DPPs and categorized them into the categories by Oumaziz *et al.* [9]. The two authors classified in the same way 89.7% of the inspected repositories, with a Cohen's kappa coefficient of 0.85, which demonstrates an almost perfect agreement [16].

Table I shows the distribution in the types of DPPs. Throughout the categorization process, we found a category that cannot be classified into the previous study's categories. We explain the inspection results and the new category below.

TABLE I: Diffusion of DPP patterns

Pattern	# (%)
Template processor	68 (68%)
Find and replace	20 (20%)
Generator	7 (7.0%)
Container build	5 (5.0%)

Out of all the repositories using DPPs, 68% repositories can be categorized into Template processors, 20% are Find and replace, and 7.0% are Generators. Compared to the findings of the previous study, the percentage of Template processor patterns is larger (this study: 68%, the previous study [9]: 54%). This might be because 15 projects changed the type of DPP to a Template processor. For example, `monicahq` projects[4] change the type of DPP from Find and replace to Template processor in May, 2020.[5]

However, we found 5 DPPs that cannot be categorized into any of the previous study's categories [9]. This is because the discovered DPPs do not generate multiple Dockerfiles (*i.e.,* the previous study investigated code clones of Dockerfiles). This study calls them the "Container build" pattern, which accounts for 5% of all the DPPs.

"Container build" pattern creates images with docker build commands, such as `docker build` and `docker buildx`. These commands have options such as `--build-arg` that can take inputs as the name of base images, versions, and architectures. Listing 6 shows an example script to realize this pattern. This script uses `docker buildx` and provides the build command with options for the CPU architecture with `--platform` (*i.e.,* line 2), image versions, and Java runtime environment version with `--build-arg` (*i.e.,* lines 3-6).

[4]https://github.com/monicahq/docker

[5]https://github.com/monicahq/docker/commit/257d3334

Listing 6: Example of DPP scripts using buildx to correspond multi images

```
1 docker buildx build
2   --platform="${DOCKER_PLATFORMS}"
3   --build-arg agent_version=
4       "$AGENT_VERSION"
5   --build-arg jre_version=
6       "$JRE_VERSION"
7   -t "${DOCKER_IMAGE}:${DOCKER_IMAGE_TAG}
     "
```

https://github.com/newrelic/infrastructure-bundle/blob/1d21/docker-build.sh

Answer to RQ0: *We discovered the "Container build" pattern, which is a unique type of DPP that does not generate multiple Dockerfiles.*

B. RQ_1: What are the characteristics of projects using DPP?

Motivation. Several studies have analyzed the activities of Docker image development, investigating and the commit activities, the release frequency of Docker images, etc. [17]–[19]. However, these studies do not distinguish whether they adopt DPPs and it is not clear what projects are more likely to adopt DPPs. In response to the first research question, we analyzed the characteristics of the projects using DPPs, comparing them with the projects that do not use DPPs.

Approach. To study the characteristics of projects that use DPPs, we compare the projects that use DPPs (hereafter, DPP projects) and the projects that do not use DPPs (hereafter, non-DPP projects). Specifically, we extract 8 metrics categorized into Snapshot metrics and Longitudinal metrics for the last year from their last commit. The definition of each metric and measurement approach are introduced below.

Snapshot metrics. The last three metrics are extracted from the latest revision of the images. Specifically, we collect these metrics from library definition files.[6] These files are plain text files contained in the `library` directory of the official-images repository. Each file is to manage the set of images that are supported by each project. The supported images written in the file appear on the Docker Hub. The file contains the URL of the Dockerfile development repository, the tags, and the supported CPU architectures of the host machines. For example of Listing 7, line 2 shows the repository URL, lines 5 and 10 describe variations of the images (*i.e.,* supported versions and platform types) using tags, and lines 6 and 11 represent the supported architectures, From this file, we retrieve the following three metrics:

1) *# Supported images* represents the number of the images that the projects support currently. We count the number of the sections that start from "Tags:" to the line with "Directory:".
2) *# Tags* represents the number of variations of docker images. In many cases, tags are used to show what base images are used in the image (*e.g.,* 3.11-alpine3.18). To measure this metric, we obtain the line beginning with "Tags:" from the latest image, separate the lines with a comma, and count the separated tags.
3) *# Architectures* represents the number of CPU architectures that the project supports. We identify the line starting "Architectures:" in the latest image, split, and count them.

[6]https://github.com/docker-library/official-images?tab=readme-ov-file#library-definition-files

Listing 7: Example of library definition file

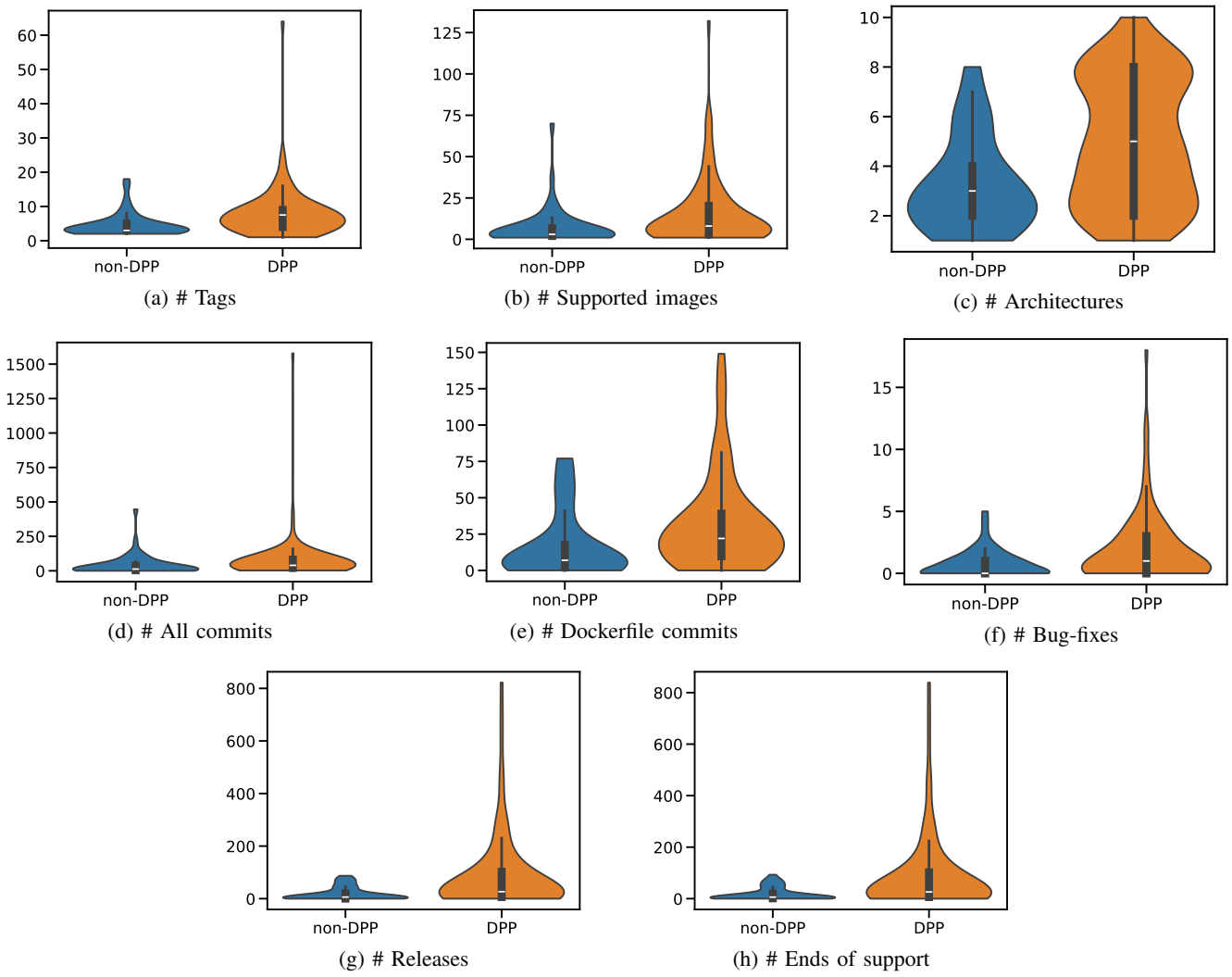
```
1 Maintainers: ..
2 GitRepo: https://github.com/docker-
   library/python.git
3 Builder: buildkit
4
5 Tags: 3.13.0a3-bookworm, 3.13-rc-bookworm
6 Architectures: amd64, arm32v5..
7 GitCommit: f064da21..
8 Directory: 3.13-rc/bookworm
9
10 Tags: 3.13.0a3-slim-bookworm..
11 Architectures: amd64, arm32v5..
```

Longitudinal Metrics. This type of metric shows how the project develops the docker images. To measure the metrics, we cloned each repository and analyzed the commit history. The definition of each metrics is described as follows.

1) *# Commits* represents the number of commits provided to the repository, disregarding what files are changed.
2) *# Dockerfile commits* represents the number of commits that changed Dockerfiles.
3) *# Bug-fixes* represents the number of commits that fixed bugs. We analyzed the commit messages and identified messages that contain "Closed", "Fixed", and similar expressions shown in the previous studies [20].
4) *# Releases* indicates the number of new images that were released. In the Docker official image repository, when a new image is developed, the library definition file needs to be updated to release the image (*i.e.,* one section of release information is inserted). To measure the metrics, we obtain the commits in the repository, identify the inserted sections of the library definition file (*e.g.,* lines 5-8 in Listing 7), and count them.
5) *# Ends of support* indicates the number of images that are no longer supported. In contrast to # Releases, we identify deleted sections of the library definition file and count them to measure the metrics.

Results. RQ1 compares the metrics between projects that adopt DPPs and projects that do not adopt DPPs. Throughout the comparison, we observe many differences between these two groups, and in most cases, larger metrics are observed in DPP projects. It is worth noting that this difference would not be due to the quality of images. This is because even the non-DPP projects that we analyzed were also nominated as Docker official images, which are admitted as high-quality images. The details of the findings are described as follows.

Findings 1: A statistically significant difference is observed between DPP-projects and non-DPP projects in



(a) # Tags (b) # Supported images (c) # Architectures

(d) # All commits (e) # Dockerfile commits (f) # Bug-fixes

(g) # Releases (h) # Ends of support

Fig. 3: Distribution of each metric in DPP projects and non-DPP projects

terms of the number of tags, supported images, and architectures. Figure 3a, 3b, and 3c show the number of tags, architectures, and supported images, respectively. In these violin plots, the thickness of the outer layer (*i.e.,* colored ones) represents the probability density of the plotted values. In the center of each violin plot, the dark box represents the interquartile range and the white dot shows the median value. The blue violin plots show each metric in non-DPP projects while orange violin plots depict those of DPP-projects.

Overall, DPP projects have more tags, supported images, and architectures. Specifically, as for the number of tags, the median in DPP projects is 7.5, which is higher than that of non-DPP projects (*i.e.,* 5.0). Similarly, the median number of supported images and architectures in DPP projects are 8.0 and 5.0, which are greater than those of non-DPP projects (*i.e.,* 5.0 and 4.0). We also applied a Mann-Whitney U test with the significance level (α) 0.01 and observed statistically significant differences for all tags ($p = 2.4e^{-5}$), supported images ($p = 0.00019$), and architectures ($p = 0.00014$).

Projects using DPPs modify Dockerfiles more frequently than projects without DPPs. Figure 3d and 3e show the number of commits, and Dockerfile commits. As for the number of commits, non-DPP projects made a median of 12.5 commits while DPP projects pushed a median of 39.5 commits. For Dockerfile commits, non-DPP projects made a median of 7.0 changes while DPP projects did a median of 22.5 commits. The Mann-Whitney U-test with the significance level (α) 0.01 identified a significant difference in the number of commits modifying Dockerfiles (All commits: $p = 4.5e^{-5}$, Dockerfile commits: $p = 0.0012$). These results show that projects using DPP update or create more Dockerfiles than non-DPP projects.

Findings 2: DPP projects release images and end the support of the images more frequently than non-DPP projects. Figure 3g and 3h show the number of releases and ends of support for non-DPP projects and DPP projects. Both the median numbers of releases and ends of support images in DPP projects are higher than in non-DPP projects. The Mann-Whitney U-test (significance level 0.01) shows a significant

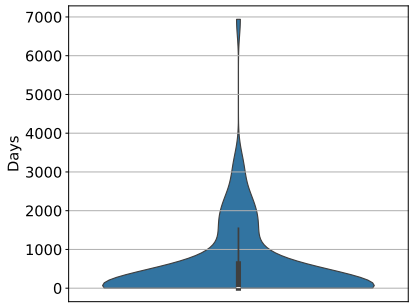


Fig. 4: Number of days between the creation of the repository and the introduction of DPP files

difference in the two groups (Releases: $p = 0.00015$, Ends of support: $p = 0.00018$). These results indicate that DPP projects more frequently released their images but at the same time abolished their older versions.

Findings 3: DPP projects fix more bugs than non-DPP projects but no statistically significant difference is observed. Figure 3f shows the distribution of bug-fixing commits relevant to Dockerfile. The median of the bug fixing commit in DPP projects (*i.e.,* 1.0) is higher than in non-DPP projects(*i.e.,* 0). Even when examining the percentage of bug fixes out of all the commits, the medians of DPP projects are still higher (DPP: 1.62%, non-DPP: 0.0%). However, the Mann-Whitney U-test with the significance level 0.01 detects no significant difference ($p = 0.035$).

Answer to RQ1: *Projects using DPPs have more tags, supported Docker images, and architecture supports than projects without DPPs. Additionally, projects using DPPs develop more actively than projects without DPPs in terms of commits, Dockerfile changes, image releases, end the support of images.*

C. RQ_2: How many projects adopt DPPs in the middle of the development?

Motivation. Oumaziz *et al.* [9] revealed that 66% of their studied repositories have DPPs and there are no off-the-shelf tools for this purpose. While they revealed the population of the projects using DPPs, it is not clear when they developed DPPs. If many projects developed DPPs in the middle of development, this may have been due to encountering difficulties. We would like to investigate to what extent projects develop DPPs in the middle of development.

Approach. In order to identify the commit that introduced DPPs, we employ `git blame` to the files constituting DPPs that we identified during the manual inspection. Note that, if the creation dates are different in the files constituting DPPs (*i.e.,* sometimes developers split files for the sake of refactoring), we use the oldest date.

Results. We describe the findings found in RQ2 as follows.

Findings 4: The majority (66%) of the studied projects adopted DPP in the middle of the development. Figure 4 shows the number of days between the creation of the repository and the introduction of DPP files. The line in the box plots indicates the median days, which is 14.5. Also, we found that 34 projects (34%) introduced DPPs on the day of the creation of the repository. In other words, 66% of the projects adopted the DPP in the middle of development.

Findings 5: 57% and 75% of projects have multiple tags and supported images, respectively. Similar to RQ1, we investigated how many images, tags, and architectures the projects had when they adopted DPPs. The median numbers of tags, supported images, and architectures are 2.0, 3.0, and 1.0, respectively. Only 33% of the projects have multiple architectures while 57% and 75% of projects have multiple tags and # Supported images, respectively. The motivation for adopting DPPs might be due to handling the number of # Supported images or tags instead of architectures.

Answer to RQ2: *The majority of the studied projects adopted DPP in the middle of the development. When they employed DPPs, more than half of the projects had already had multiple Supported images and Tags.*

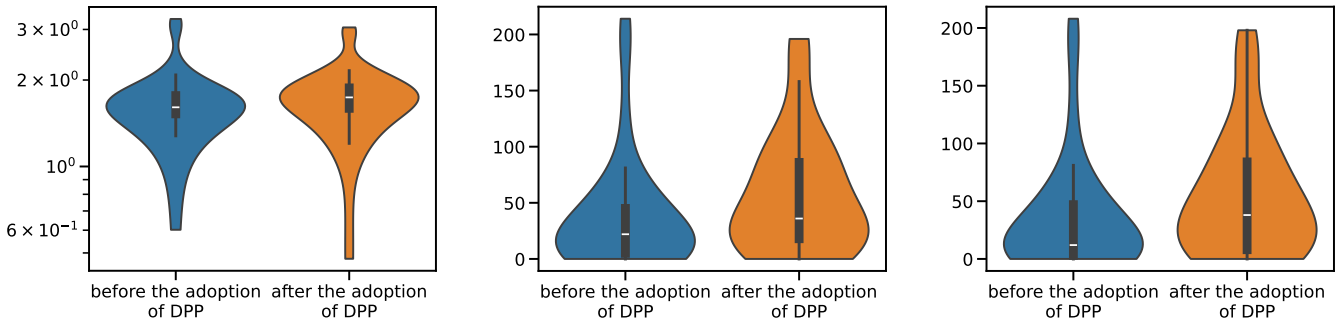
D. RQ_3: What purposes do developers adopt DPP?

Motivation. Oumaziz *et al.* conducted interviews with developers from 25 projects to study whether developers use DPPs and, if they do, what is their level of satisfaction. Yet, it is not clear what are the purposes or motivations for adopting DPPs. In RQ3, we would like to know the purpose of adopting DPP.

Approach. To investigate the purpose of developing DPPs, we manually investigate the messages of the commits that introduced DPPs. We first clone the repositories where DPPs are developed in the middle of the development. We then identify commits that introduced DPPs by applying `git blame` to DPP files that are found by our manual identification process (*i.e.,* Section III). We finally manually examine the commit messages and the associated GitHub issues in order to classify the purposes. Specifically, two of the authors independently inspected each of the code reviews and annotated the purpose. In 70.3% of the inspected cases, the two authors reached an agreement, which resulted in a Cohen's kappa coefficient of 0.77, demonstrating a substantial agreement [16]. To resolve the disagreements, a third author inspected the conflicting case with the corresponding annotations and then recommended a final annotation. The discussions with the three inspectors were done until a full agreement was reached. The three annotators have 2, 7, and 9 years of docker programming experience.

Results. Table II summarizes the reasons for adopting DPP. Note that 16 commit messages do not have enough information to determine the labels (*i.e., "added update script"*) so we categorize them into "Not enough information".

Findings 6: There are various reasons for adopting DPPs in the middle of the development. Throughout our manual inspection, we observed 14 categories. The most common category was "Automate creating Dockerfiles" and we observed 15 cases that fell into this category. We found that many developers want to generate Dockerfiles in order to reduce the effort to write Dockerfiles. For example, a commit



(a) All commits (log scale) (b) # Releases (c) # Ends of support

Fig. 5: Distribution of each metric before/after 1 year since DPP adoption

TABLE II: Reasons for DPP adoption

Reason	Number of projects
Automate creating Dockerfiles	15
Easier updates for multi versions	9
Easier updates for multi base image	9
Easier updates for multi architecture	5
Easier updates shasums	4
Make testing easier	2
Others	6
Not enough information to classify	16
Total	66

message indicates "Allows simple automation of almost all project operations."

"Easier updates for multi versions" (9 cases) was the second most common category, followed by "Easier updates for multi variants" (9 cases), and then "Easier updates for multi architectures" (6 cases). In an example of "Easier updates for multi versions", the commit message indicates "Add simple update.sh script to make version bumps easier." Notably, in contrast to our expectation, we observed only 2 cases of "make testing easier," and 1 case of "avoid error," (in the "Other" category). This implies that DPPs are developed to reduce the effort to create or update Dockerfiles instead of reducing the risk of introducing bugs.

Answer to RQ3: *The most common reason for adopting DPP is to simply automate their tasks. Additionally, DPP is developed to support multiple variants, versions, and architectures of images.*

E. RQ_4: Do adopting DPPs impact release activities?

Motivation. DPPs can automatically generate various Docker images, which would facilitate release activities. We thus hypothesize that adopting DPP increases the frequency of releasing or types of tags and architectures. Inspired by previous studies examining the activities changes [21]–[23], we decided to know the difference before and after adopting DPPs.

Approach. To examine the impacts of adopting DPPs, we compare the development activities for one year before and after the DPP being adopted, inspired by previous studies [21]–[23]. From these two periods, we measure 8 metrics used in RQ1, compare them, and apply the Mann-Whitney U test with the significance level 0.01 to confirm the statistically significant difference.

Results. Figure 5a, 5b and 5c depict the number of each before and after DPP adoption. Note that, due to the limited space, we present the number of commits, releases, and ends of supports in the figure but that of the other metrics can be seen on our replication package.[7]

Findings 7: The median number of # Releases and # Ends of support after the adoption of DDPs are larger than before but no statistically significant difference is observed. The median number of commits, # Releases, and # Ends of support after the adoption of DDPs are larger than before. When applying the Mann-Whitney U test with the significance level (α) 0.01, we observe no statistically significant difference (Releases: $p = 0.072$, Ends of support: $p = 0.066$). As for other metrics, there are no changes before and after the adoption of DPPs as well as no statistically significant differences.

In RQ3, our manual inspection revealed that the second and third major reasons for adopting DPPs are to deal with multiple versions or architectures. However, RQ4 shows that the release activities have not significantly changed. Thus, DPPs might help only reduce the effort of creating Dockerfiles.

Answer to RQ4: *In all the metrics, we do not observe statistically significant differences before and after the adoption of DPPs. The adoption of DPP reduces the effort of creating Dockerfiles but might not facilitate release or commit activities.*

F. RQ_5: How frequently do developers maintain DPPs

Motivation. While DPPs may help generate Docker images, developers may need to maintain DPPs. However, it is unclear how much effort developers need to maintain DPPs. This finding would provide practitioners with the materials in order to decide on developing DPPs. To understand the effort of managing DPPs, RQ5 studies the frequency of commits to modify DPPs and the size of changes.

Approach. RQ5 examines how frequently developers maintain DPPs after adopting DPPs. Specifically, we measure the ratio of commits that change DPPs out of all the commits that

[7]https://anonymous.4open.science/r/scam2024_replication_package-2086

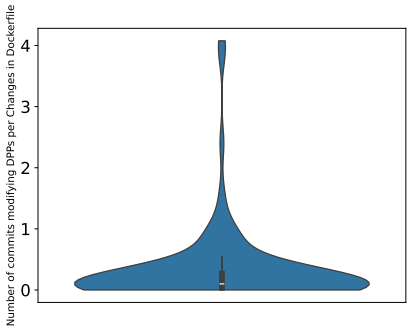


Fig. 6: Number of DPP commits per a commit that modifies Dockerfiles

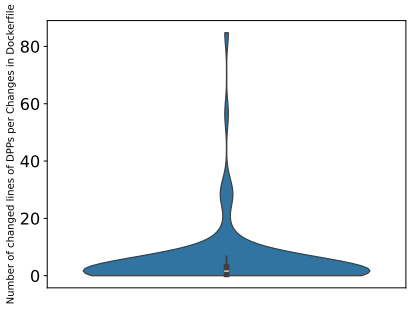


Fig. 7: Number of changed lines in DPP files per a commit that modifies Dockerfiles

change Dockerfiles. The reason why we use the ratio is because the period and activeness of the development differ across projects. To determine which files constitute DPPs, we use the manual inspection results in RQ0. Also, we measured the lines of changes in DPPs as a proxy metric of the effort.

Note that we except data files (*i.e.,* text files or json files) and not pure DPP files such as build files (*e.g.,* Makefile) from the calculation because the first one is not programming and the second one is difficult to isolate the part of DPPs from other build parts. Additionally, the calculations include all the DPPs (*i.e.,* disregarding when DPPs are developed, at the beginning or in the middle of the development).

Results. We describe the findings found in RQ5 as follows.

Findings 8: Developers make a change in DPPs per 10 commits modifying Dockerfiles. Figure 6 shows the number of DPP commits per a commit that modifies Dockerfiles. As a result of measuring the ratio, we found that a median of 0.10 changes and a mean of 0.31 changes to DPPs are made per a change in Dockerfiles. The quartile range is from 0.03 to 0.27, so developers often update DPPs. The minimum ratio is 0.00, implying that some projects do not need to update DPPs.

Also, we looked into the size of changes in DPPs. Figure 7 shows the changed lines per a commit modifying DPP. We can see that the variance is large. However, the median is 1.61, and the quartile range is from 0.45 to 3.08, which is not large. This implies that developers update DPPs frequently, but make small changes.

Findings 9: The "Template processor" patterns require less efforts than the "Find and replace" pattern. To investigate which type of DPPs require less effort to update DPPs, we also measure the above ratio for "Template processor" and "Find and replace". We do not show the other categories because the samples of other categories are not sufficient.

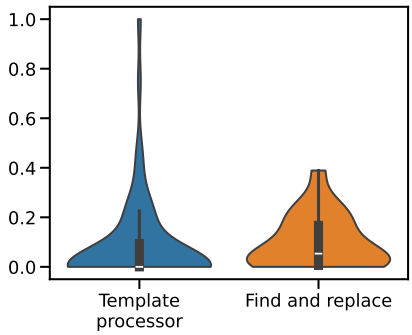


Fig. 8: Ratio of DPP commits of each DPP type

Figure 8 shows the ratio of the commits modifying DPPs and Dockerfile. The median ratio regarding "Find and replace" DPPs are larger than "Template processor" DPPs (*i.e.,* 0.05 and 0.0, respectively). This might be because many projects using "Find and replace" need to manually update version numbers (e.g., "10.0.1") written in the DPPs. On the other hand, projects using "Template processor" automate not only generating Dockerfiles but also obtaining the version number of images, resulting in fewer updates of DPP files.

Answer to RQ5. Developers need the maintenance effort for updating DPPs after the adoption, specifically, developers need a change in DPPs per 10 commits modifying Dockerfiles.

V. Summary and Implication

In this study, we identified projects using DPPs throughout manual inspections. We investigated the characteristics of projects using DPPs, the timing, motivation, impact of adopting DPPs, and the effort of maintaining DPPs. Our empirical results show that (i) there is the "Container build" pattern that does not generate multiple Dockerfiles; (ii) Projects using DPPs have more tags, supported Docker images, and architecture supports than projects without DPPs; (iii) 66% of projects develop DPPs in the middle of development; (iv) the most common reasons for adopting DPP is to reduce the effort of creating Dockerfiles, and to ease updating versions/variations/architectures; (v) the adoption of DPPs do not facilitate development activities.

Takeaways for researchers: In RQ0, we discovered the "Container build" categories that do not generate multiple Dockerfiles and also found that some projects switch the types of DPPs. Also, RQ3 shows that projects developed DPPs with different purposes, *e.g.,* automation to create Dockerfiles, and easier updates for multiple versions, variations, and architectures. However, it is still unclear what types of DPPs are the most helpful for each aim. Thus, *researchers should evaluate each type of DPP in terms of the efficiency and maintainability of dealing with Dockerfiles*.

Takeaways for practitioners: Although DPPs have many benefits, they also have negative aspects. To keep using DPPs, effort is required to maintain them. In fact, as RQ5 shows, projects are required to modify DPPs about 1 time per 10 changes (*i.e.,* commits) in Dockerfiles. Therefore, *practitioners should be aware of the maintenance costs of DPPs in addition to developing DPPs for the first time*.

VI. Threat to Validity

Our study is subject to several threats to validity.

Threats to internal validity: We manually classified projects with/without DPPs. We carefully inspected the files contained in the repositories but may have missed DPPs. To mitigate this threat, two inspectors independently examined the files and also asked a professional who is familiar with Docker development to classify them.

Threats to construct validity: We extract several metrics from the latest revisions and histories to investigate the impact of adopting DPPs (RQ4). Contrary to our hypothesis, we observe no differences before and after DPPs. However, RQ3 finds the clear purposes of developing DPPs such as handling many variations of images. The impact might be seen by using the other metrics. Our future study needs expanding metrics.

Threats to external validity: This study examined only the Docker official repositories because inspecting all the files contained in the repositories takes a lot of time. In order to increase the generality of our findings, we are planning to extend the dataset by inspecting other repositories such as those owned by verified publishers.[8]

VII. Related work

This section introduces our related work.

A. Docker images and Dockerfiles

Cito *et al.* [2] examined 70,197 Dockerfiles to characterize the Docker ecosystem. They showed that most containers inherit base images of heavy-weight operating systems. Also, they discovered that 28.6% of issues arise from missing version pinning. Lin *et al.* [18] also analyzed 3,364,529 Docker images and their repositories to characterize the Docker ecosystems. They found that there are more images that inherit language run-time images or application images than those that inherit OS images directly. Additionally, the number of images using the ARM architecture is also increasing significantly.

Ye *et al.* [24] proposed a tool to create Dockerfiles according to the software to be installed in a Docker container. They showed that the tool can generate the desired Dockerfile with 59% accuracy. Hassan *et al.* [25] has proposed RUDSEA, an approach to recommend updates of dockerfiles, based on analyzing changes in software environments and their impacts.

Gholami *et al.* [26] studied the official Docker images on Docker Hub and explored how packages are changed in these images. They found that a median of 8.6 packages are upgraded in their images. Tanaka *et al.* [19] investigated the maintenance in Dockerfile update activities. They showed that, 7,914 of the repositories containing Dockerfiles with version-pinned packages, 18% had version updates, 37% had no version updates, and the remaining 45% are dormant repositories.

Eng *et al.* [17] investigated how instructions and base images are used in Dockerfiles over time. Their empirical investigation showed that more lightweight images are being used as opposed to OS images. Wu *et al.* [27] examined 4,110 repositories containing Dockerfiles to investigate what instructions are frequently updated in Dockerfiles as well as study the co-changed files. They showed that RUN instructions are more frequently changed than the others.

Many previous studies investigate updates in Dockerfile, but few studies examine DPPs and their impact.

B. Code Preprocessors

Ernst *et al.* [28] conducted an empirical investigation of 26 packages comprising 1.4 million lines of real-world C code to examine how the C preprocessor is used in practice and classified the purposes of C preprocessor. Mazinanian *et al.* [29] investigated 150 CSS preprocessors to reveal the usage patterns of their four features. They showed that developers have a clear preference for global variables, especially variables storing color values.

Ye *et al.* [24] have developed DockerGen, which containerizes the target software. DockerGen creates a Dockerfile, selecting the base image, dependencies, etc. They showed that 73% of the created Dockerfiles can be built successfully on 100 software packages of various categories. Eric *et al.* [30] have developed Dockerizeme, which is a knowledge-based approach to infer the dependencies needed to execute a Python code snippet without import error. They observed that Dockerizeme can resolve import errors in 892 out of about 3,000 GitHub gists.

The previous studies examine the preprocessors in many programming languages but few studies investigate the preprocessor of Dockerfiles for multiple Docker environments.

VIII. Conclusions

This study explored the characteristics of projects using Dockerfile Preprocessors, the timing, impact, and aim of adopting Dockerfile Preprocessors, and the maintenance effort. Our empirical results demonstrated that (i) there is the "Container build" pattern that does not generate multiple Dockerfiles; (ii) Projects using DPPs have more tags, supported Docker images, and architecture supports than projects without DPPs; (iii) 66% of projects develop DPPs in the middle of development; (iv) the most common reasons for adopting DPP is to reduce the effort of creating Dockerfiles, and to ease updating versions/variations/architectures; (v) the adoption of DPPs do not facilitate development activities.

Acknowledgment

We gratefully acknowledge the financial support of JSPS for the KAKENHI grants (JP21H03416, JP24K02921, JP24K02923), Bilateral Program grant JPJSBP120239929, and JST for the PRESTO grant JPMJPR22P3.

[8] https://hub.docker.com/search?image_filter=store

REFERENCES

[1] Y. Zhang, B. Vasilescu, H. Wang, and V. Filkov, "One size does not fit all: an empirical study of containerized continuous deployment workflows," in *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE)*, 2018, pp. 295–306.

[2] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall, "An empirical analysis of the docker container ecosystem on github," in *Proceedings of IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 323–333.

[3] Portworx, "Annual container adoption report," 2019, https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf (Accessed at 2023-01-31).

[4] M. U. Haque, L. H. Iwaya, and M. A. Babar, "Challenges in docker development: A large-scale study using stack overflow," in *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2020, pp. 1–11.

[5] D. Morris, S. Voutsinas, N. C. Hambly, and R. G. Mann, "Use of docker for deployment and testing of astronomy software," *Astronomy and computing*, vol. 20, pp. 105–119, 2017.

[6] "Docker hub," https://hub.docker.com/ (Accessed at 2024-06-01).

[7] Y. Zhang, G. Yin, T. Wang, Y. Yu, and H. Wang, "An insight into the impact of dockerfile evolutionary trajectories on quality and latency," in *Proceedings of the IEEE 42nd annual Computer Software and Applications Conference (COMPSAC)*, vol. 01, 2018, pp. 138–143.

[8] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. S. Warke, M. Mohamed, and A. R. Butt, "Large-scale analysis of the docker hub dataset," in *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2019, pp. 1–10.

[9] M. A. Oumaziz, J.-R. Falleri, X. Blanc, T. F. Bissyandé, and J. Klein, "Handling duplicates in dockerfiles families: Learning from experts," in *Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 524–535.

[10] D. Merkel *et al.*, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 239, no. 2, p. 2, 2014.

[11] A. Acharya, J. Fanguède, M. Paolino, and D. Raho, "A performance benchmarking analysis of hypervisors containers and unikernels on armv8 and x86 cpus," in *Proceedings of European Conference on Networks and Communications (EuCNC)*. IEEE, 2018, pp. 282–9.

[12] S. Shirinbab, L. Lundberg, and E. Casalicchio, "Performance evaluation of container and virtual machine running cassandra workload," in *Proceedings of 3rd International Conference of Cloud Computing Technologies and Applications (CloudTech)*. IEEE, 2017, pp. 1–8.

[13] "Docker overview," https://docs.docker.com/guides/docker-overview/ (Accessed at 2024-06-01).

[14] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps, "A dataset of dockerfiles," in *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, 2020, p. 528–532.

[15] "Docker official images," https://docs.docker.com/trusted-content/official-images/ (Accessed at 2024-06-01).

[16] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[17] K. Eng and A. Hindle, "Revisiting dockerfiles in open source software over time," in *Proceedings of IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 449–459.

[18] C. Lin, S. Nadi, and H. Khazaei, "A large-scale data set and an empirical study of docker images hosted on docker hub," in *Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 371–381.

[19] T. Tanaka, H. Hata, B. Chinthanet, R. G. Kula, and K. Matsumoto, "Meta-maintanance for dockerfiles: Are we there yet?" *arXiv preprint arXiv:2305.03251*, 2023.

[20] V. Lenarduzzi, F. Palomba, D. Taibi, and D. A. Tamburri, "Openszz: A free, open-source, web-accessible implementation of the szz algorithm," in *Proceedings of the 28th international conference on program comprehension (ICPC)*, 2020, pp. 446–450.

[21] O. Nourry, Y. Kashiwa, Y. Kamei, and N. Ubayashi, "Does shortening the release cycle affect refactoring activities: A case study of the jdt core, platform swt, and ui projects," *Information and Software Technology*, vol. 139, no. C, 2021.

[22] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016, pp. 858–870.

[23] D. A. d. Costa, S. McIntosh, C. Treude, U. Kulesza, and A. E. Hassan, "The impact of rapid release cycles on the integration delay of fixed issues," *Empirical Software Engineering*, vol. 23, pp. 835–904, 2018.

[24] H. Ye, J. Zhou, W. Chen, J. Zhu, G. Wu, and J. Wei, "Dockergen: A knowledge graph based approach for software containerization," in *Proceedings of the IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2021, pp. 986–991.

[25] F. Hassan, R. Rodriguez, and X. Wang, "Rudsea: Recommending updates of dockerfiles via software environment analysis," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018, p. 796—801.

[26] S. Gholami, H. Khazaei, and C.-P. Bezemer, "Should you upgrade official docker hub images in production environments?" in *Proceedings of IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2021, pp. 101–105.

[27] Y. Wu, Y. Zhang, T. Wang, and H. Wang, "Dockerfile changes in practice: A large-scale empirical study of 4,110 projects on github," in *Proceedings of the 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2020, pp. 247–256.

[28] M. D. Ernst, G. J. Badros, and D. Notkin, "An empirical analysis of c preprocessor use," *Software Engineering*, vol. 28, no. 12, p. 1146—1170, 2002.

[29] D. Mazinanian and N. Tsantalis, "An empirical study on the use of css preprocessors," in *Proceedings of IEEE 23rd international conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 168–178.

[30] E. Horton and C. Parnin, "Dockerizeme: Automatic inference of environment dependencies for python code snippets," in *Proceedings of IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 328–338.