Handling Duplicates in Dockerfiles Families: Learning from Experts

Mohamed Oumaziz, Jean-Rémy Falleri, Xavier Blanc, Tegawendé Bissyandé, Jacques Klein

Handling duplicates in Dockerfiles families: Learning from experts

Mohamed A. Oumaziz, Jean-Rémy Falleri, Xavier Blanc
University of Bordeaux
LaBRI, UMR 5800
F-33400, Talence, France
Email: {moumaziz, falleri, xblanc}@labri.fr

Tegawendé F. Bissyandé, Jacques Klein
University of Luxembourg
SnT Interdisciplinary Center
L-1855, Luxembourg
Email: {tegawende.bissyande, jacques.klein}@uni.lu

***Abstract*—Docker is becoming a popular tool used by developers and end-users to deploy and run software applications. Dockerfiles are now found alongside projects' source code. Several projects are even starting to maintain families of Dockerfiles, like the Python project that maintains a family of 43 Dockerfiles, each for a specific Python version on a specific Linux distribution. In some situations, Dockerfiles family maintainers have to propagate a change to several, if not all, Dockerfiles of the family (for instance a bugfix applying on all Dockerfiles targeting Python 2.7). This need to propagate changes is usually due to the presence of duplicates between several Dockerfiles of the family. In this paper, our goal is to provide practitioners a clear explanation for why Dockerfile duplicates arise in projects, and what are the different means to handle duplicates with their pros and cons. To perform a grounded analysis, we observe the practices of expert Dockerfile maintainers of Official Docker projects, which are setting the best-practices for Dockerfile maintenance. We show that duplicates in Dockerfiles are frequent in our corpus, that maintainers are aware of their existence, are frequently facing them and have a mixed opinion regarding them (error-prone when not using any tool, but easy to maintain with the right tools). Finally, we describe and analyse the tools used by maintainers to handle duplicates.**

***Index Terms*—Dockerfile, Duplicates, Empirical study, Software reuse**

I. Introduction

Docker simplifies the deployment of software applications by enabling developers to package their applications with all their required dependencies in a single binary unit, called a *Docker image*[1]. To build an image, developers have to write a *Dockerfile*, that will be executed by Docker. Such a Dockerfile is written in a Domain Specific Language (DSL) that can be compared to a very limited shell scripting language.

Software projects using Docker manage their Dockerfiles in their Software Configuration Management tool (e.g. GitHub), as any other software artefacts. Some projects even have to manage a family of Dockerfiles (DF for short), especially when they have to support and maintain several versions in parallel with several sets of dependencies. For instance, the Python project[2] provides a family of 43 Dockerfiles, each one targeting a specific version of python (e.g. 2.7, 3.8-rc) with a specific set of distributions (e.g. Debian, Alpine, Windows).

Looking at such projects, it appears that sometimes a patch targeting a Dockerfile has to be propagated to some (if not all) Dockerfiles of the family. As an example, Listing 1 shows a patch encountered in the official Python project[3]. This patch fixes an unexpected segmentation fault that appears only on Dockerfiles using the Alpine Linux distribution. This bug is due to a too small stack size and is resolved by adding a new flag in a *make* command as shown in Listing 1 (line 5). This bugfix was propagated to the 7 Dockerfiles of the family that are based on the Alpine distribution.

Listing 1. A bugfix of the same duplicate across the Alpine family of Dockerfiles in the official Python project

```
1  RUN ...
2  && apk add --no-cache --virtual .build-deps  \
3  bzip2-dev \
4  ...
5  && make -j "$(nproc)"
        EXTRA_CFLAGS="-DTHREAD_STACK_SIZE=0x100000"
6  && ...
```

As another example, every image of the official Python project come with a python package management tool called *pip*. When a new version of *pip* is available, all the family's Dockerfiles have to be patched. We observed such situation when an update was done from pip version 19.0.2 to 19.0.3[4] as shown in Listing 2.

Listing 2. An extract of the dicussed version number update across the whole Dockerfile family

```
ENV PYTHON_PIP_VERSION ~~19.0.2~~ 19.0.3
```

The two previous examples highlight a case of patch propagation among a family of Dockerfiles. Such situations arise because the impacted Dockerfiles share duplicated instructions. In the first example, all the Dockerfiles that are based on the Alpine distribution share the same *make* command with the same arguments (see Listing 1 (line 5)). In the second example, all the Dockerfiles mention the pip version they depend on. As a consequence, for a patch to be safely propagated, all the impacted duplicates have hence to be identified and then equally modified, which is time consuming and may be error-

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

[2] https://hub.docker.com/_/python/

[3] https://github.com/docker-library/python/commit/8717dc2523c8093990cb

[4] See the 4 commits performed the 04.20.2019 (i.e. https://github.com/docker-library/python/commit/af2cf72d9c6c304d041c88db3)

prone. For example, this patch applied in 2015[5] has required another commit the same day to propagate the modification to other Dockerfiles[6].

The main questions are then: are duplicates abundant in Dockerfiles families? and if yes, is their handling a source of concerns? Such questions are relevant to any maintainers of Software projects that rely on Docker, and that come to manage several Dockerfiles.

In this paper, we answer the question of duplicates handling in Dockerfiles families by providing a grounded study based on the analysis of the *Official Docker Projects*. This choice is driven by the fact that these projects are real-world popular projects that manage medium to large families of Dockerfiles, but most importantly because as it is written in the Docker documentation, these projects promote the best practices to maintain Dockerfiles[7] and may be considered as a reference for any Dockerfile maintainer.

Our hypothesis is that these projects are the best to highlight the duplicates in Dockerfiles families problem, and the best to define the cutting-edge practices that exist for handling them.

In our study, we then follow a mixed research method (quantitative and qualitative) and more precisely answer the following research questions:

1) Do official projects maintain families of Dockerfiles, and why?
2) Do duplicates arise in Dockerfiles families and why?
3) What are the tools used by official Docker projects for managing duplicates? What are their pros and cons?

Through our study, we show that official Docker projects frequently maintain families of Dockerfiles. We observe that the most common reasons are: supporting multiple versions/base-images and versions/flavours. We then show that duplicates in Dockerfiles are abundant, and find the underlying reasons behind them: Software installation and configuration, dependency management and runtime configuration. We also perform a survey on Dockerfile families maintainers (DFM in short) of official projects and find out that they are aware of the duplicates existence and are frequently facing them. However, while these duplicates are abundant, DFM have a mixed opinion regarding them. Some of them do not use specific tools for handling duplicates, and state that their handling may be error-prone. Others use specific tools for handling them, and state that they are easy to maintain. We also find that the specific tools range in the following categories: template processors, code generators, find and replace executors. We further show that projects using template processors and code generators manage to reduce the amount of duplicates with a median at 30% that goes up to 100% for generators. Finally, we find that such tools can be hard to set-up and use, calling for more research on dedicated tools to handle duplicates in Dockerfiles families.

II. Dockerfiles in a Nutshell

A *Docker container* is a "lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings"[8].

A Docker container is ran from a *Docker image*, which is built from a *Dockerfile*. Using Docker is therefore a two steps process (1) Build a *Docker image* from a *Dockerfile* (*docker build -f Dockerfile -t my_image*), (2) Run the *Docker container* from the *Docker image* (*docker run my_image*).

In our study, we focus on the first step, and more specifically on Dockerfiles, which are scripts composed of a sequence of *instructions*. Listing 3 shows a sample Dockerfile for a web application written in Python.

Listing 3. A sample Dockerfile for a hypothetical Python application

```
# Use the official python base image
FROM python:2.7-slim
# Set the working directory to /app
WORKDIR /app
# Copy current directory content into the container at /app
ADD . /app
# Install dependencies
RUN apt-get update && apt-get install -y apache2
# Make port 80 available to the world outside this container
EXPOSE 80
# Define environment variable
ENV NAME Hello World
# Run "python app.py" when the container is executed
CMD ["python", "app.py"]
```

This Dockerfile starts with a `FROM` instruction stating that the image to be built will be based on the existing *python:2.7-slim* image. The `FROM` instruction is mandatory, and several base images are provided by Docker (Ubuntu, Alpine, etc.). The Dockerfile then contains three instructions that will be executed when building the image. The `WORKDIR` instruction states the working directory of the image. The `ADD` instruction copies all the files contained in the local current directory into the */app* directory of the image. The `RUN` instruction asks docker to run some commands inside the image, aiming at modifying the image. Here, the two commands are *apt-get update* and *apt-get install -y apache2*. When Docker will build the image, it will then refresh the list of packages of the image, and then install Apache2. It should be noted that `RUN` instructions create a layer (persistent state) inside the image. Thus, increasing build time and final image size. For these reasons, Docker best practices recommend to use as few RUN instructions as possible. Therefore, commands are frequently combined inside a single `RUN` instruction using shell delimiters (such as '&&' and ';'). The three last instructions of the Dockerfile will be used by Docker when it will run the image and hence instantiate the container. The `EXPOSE` instruction asks Docker to open a port of the container (here the port 80). The `ENV` instruction asks Docker to set an environment variable

[5] https://github.com/docker-library/python/commit/f9739c6da575c450aaed8628c1e0bfa97bf1ba18

[6] https://github.com/docker-library/python/commit/00c226b82eee61c6c68adf813d9f7177d2efa52a

[7] https://docs.docker.com/docker-hub/official_images/

[8] https://www.docker.com/what-container

of the container. The Dockerfile states the default value of this environment variable, but other values can be provided as an argument of the *docker run my_image* command. Finally, the CMD instruction asks the Docker container to execute this default command just after its creation.

III. Data collection

In this section, we present our three main data sources that are used in the remainder of this paper: a list of 99 GitHub repositories each one corresponds to project that manages a DF, an on-line survey performed on 25 Dockerfile family maintainers (DFM) from our 99 repositories and 877 duplicates among the DF of our repositories. All repositories, survey questions, duplicates are available on-line[9].

A. Repositories

As presented in Section I, we choose to look at official Docker projects only. The list of these official projects is available on GitHub, with 138 official projects, as of March 2019.[10] Some of these projects share the same Github repository. For example, InfluxDB and Chronograf are maintained in the same GitHub repository. We therefore remove these 9 projects from our corpus to preserve a dataset uniformity. We also discard the *hello world* project as it is only intended to provide examples for beginners. At this stage, our corpus includes 128 projects with their associated GitHub repositories.

We then count the number of files named Dockerfile in each of these 128 projects, as it is the most common name for Dockerfiles (see Section VII for potential threats). We identify 1300 Dockerfiles. Figure 1 depicts the number of Dockerfiles maintained by each of these 128 projects. We note that half of the projects maintain at least 4 Dockerfiles with 25% maintaining more than 10 Dockerfiles. As we are interested by Dockerfiles families, we discard 29 projects maintaining a single Dockerfile. Finally, our corpus contains 99 official Docker projects.

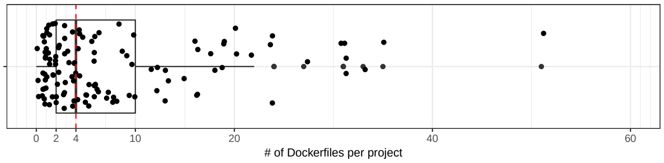


Fig. 1. Boxplot for the number of Dockerfiles maintained by each project.

B. Survey answers

Our study involves a survey of official Dockerfile maintainers (DFM) of projects in our corpus to get their opinions about duplicates and understand their practices.[11] Therefore, we create a survey composed of two main sections asking questions about 1) duplicates and 2) management tools to handle duplicates. In order to retrieve their contact information, we analysed all commits from all Github repositories of projects in our corpus and extracted the commit author's e-mail address. Using this process, we retrieved about 900 e-mail addresses. Finally, we send a link to our online survey using these addresses and gathered 29 responses, leading to a response rate of about 3% which may indicate that a large set of emails were actually from no longer active official DFM or not active at all (95 bounced emails). Out of these 29 responses, 25 maintainers were actually maintaining more than a single Dockerfile as we address in our study. The 4 remaining maintainers were ignored in these results (they left the survey directly after the first question).

```
RUN apt-get update && apt-get install -y apache2
```

↓

```
RUN apt-get update
RUN apt-get install -y apache2
```

Fig. 2. RUN instruction with multiple shell commands split into two RUN instructions, one for each shell command.

C. Duplicates

We consider a Dockerfile duplicate as a sequence of instructions that is duplicated across several Dockerfiles of the same project. We call *size* the number of instructions contained in the duplicate and *owners* the number of files containing the sequence. We provide a tool that inputs a list of git repositories and outputs a list of maximal duplicates found among each repository's Dockerfiles.

As a first step, our tool starts by scanning repositories looking for Dockerfiles and parses them. During the parsing process, we ignore all comments, we identify instructions' types, we extract the instructions' argument's text and normalise their white-spaces (replacing tabs by spaces, removing carriage returns and keeping only one space between two words) to avoid missing duplicates because of indentation differences. As said in Section II, RUN instructions often have arguments that are composed of many shell commands. Therefore, to also detect duplicates inside these shell commands, we choose to split these instructions, producing one RUN instruction for each contained shell command as seen in Figure 2. Instructions are split using the classical && and ; shell command delimiters.

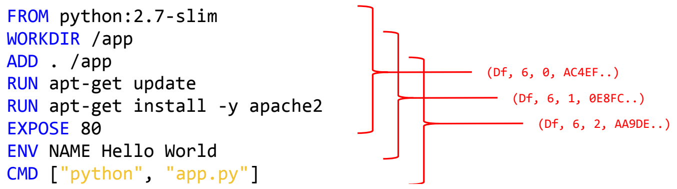
Once all Dockerfiles are parsed, we extract the maximum-sized duplicates among each project's Dockerfiles. We use the *index-based duplicate detection* technique [1]. First, for each Dockerfile in our corpus, we extract and index the hashes of all possible chunks of instructions for a given size, as shown in Figure 3. The chunks' size goes from the maximal number of instructions contained in the Dockerfile to one which is the smallest granularity: one instruction.

Figure 3 shows an example of hashes extracted with the chunk size set at 6. The tuple *(DF, 6, 0, AC4EF..)* indicates respectively: the filename, chunk size, chunk's first instruction location in the file, the chunk's hash. After having indexed all

[9] https://se.labri.fr/a/ICSME19-docker-oumaziz

[10] https://github.com/docker-library/official-images/tree/master/library

[11] https://forms.gle/sDMkcvSQxnMJwXaA6



```
FROM python:2.7-slim
WORKDIR /app
ADD . /app
RUN apt-get update
RUN apt-get install -y apache2
EXPOSE 80
ENV NAME Hello World
CMD ["python", "app.py"]
```

Fig. 3. Dockerfile presenting an example of duplicate index with chunk size set to 6.

```
1   FROM alpine:3.6
2   ENV _BASH_GPG_KEY 7C0135FB088AAF6 \
3   C66C650B9BB5869F064EA74AB
4   ENV _BASH_VERSION 3.1
5   ENV _BASH_PATCH_LEVEL 0
6   ENV _BASH_LATEST_PATCH 23
```

Fig. 4. Extract of real Dockerfile duplicate from Bash shell v3.1

hashes, we apply the algorithm described in [1] to extract the maximum-sized duplicated chunks (duplicated chunks that are not contained in an other duplicated chunk). Also, contrary to Hummel et al. [1] we don't apply any code normalisation except what was described previously.

Figure 4 and Figure 5 show two extracts of Dockerfiles from our corpus. They are part of Dockerfiles used to build images for the well known bash shell (versions 3.1 and 4.0). After applying our technique, we identify two duplicates with two different sizes. The first duplicate has a size of 2, ranging from line 1 to 3, while the second duplicate has the smallest possible size which is 1 corresponding to line 5.

Finally, we apply our tool on each repository from our list, and identify a total of 877 duplicates across all projects. These duplicates will constitute our duplicates dataset for the remainder of this study.

IV. RQ1: Do official projects maintain families of Dockerfiles, and why?

To answer our first research question, we perform a manual inspection of all Dockerfiles contained in a project. We then identify the reasons behind the existence of multiple Dockerfiles in a project.

1) Methodology: While looking at Dockerfiles when building our dataset, we note that the location path of a Dockerfile contains information about its purpose. For instance, the `rabbitmq` project has a Dockerfile located in `3.6/alpine/Dockerfile`. Based on this path, we can easily understand that this Dockerfile targets the version `3.6` of `rabbitmq` and is based upon Alpine Linux, a lightweight Linux distribution. Therefore, by analysing all Dockerfiles' paths, we see that `rabbitmq` maintains Dockerfiles for various versions and with various base images.

```
1   FROM alpine:3.6
2   ENV _BASH_GPG_KEY 7C0135FB088AAF6 \
3   C66C650B9BB5869F064EA74AB
4   ENV _BASH_VERSION 4.0
5   ENV _BASH_PATCH_LEVEL 0
6   ENV _BASH_LATEST_PATCH 44
```

Fig. 5. Extract of real Dockerfile duplicate from Bash shell v4.0

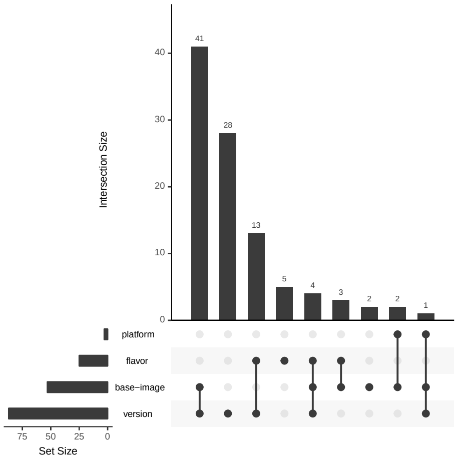


Fig. 6. UpSet plot showing the relationships between versions, flavours, base images and platforms across our repositories.

To perform this analysis on all of our repositories, we follow a semi-automated process. First, for each project, we retrieve the path of every Dockerfile. We then split all paths using the directory separator symbol. For a given nesting level `i` in the path, we establish the list of all values seen at this level. For instance for the following set of paths: `3.6/alpine/Dockerfile`, `3.7/alpine/Dockerfile` we extract the following set of names: *Level 1:* `3.6`, `3.7`, *Level 2:* `alpine`. For each project, the first author goes through the whole list of extracted names and for each of them, either creates a new category and associates the name to it, or associates the name to a category that was created for a previous name. Finally, we count the number of elements in each category.

2) Results: Based on our methodology, we identify four categories. The most represented category is *version* containing several versions of the project, similarly to the `rabbitmq` example. The second category by order of importance is *base image* it contains the different base images used to produce the project images (e.g. Alpine Linux and Debian Linux). The third category is *flavour* containing different variants of the project (e.g. normal or multi-threaded flavour for the Perl project[12]). The most seldom category is *platform*, containing the different Docker platforms supported by the project (e.g. ARM, x86).

Figure 6 is an *UpSet* plot [2] showing the relationships between versions, flavours, base images and platforms across the projects in our repositories. We first notice that the majority of projects (41) are actually maintaining images for multiple base images and versions at the same time. We also note that maintaining multiple images only because of multiple versions

[12] https://github.com/Perl/docker-perl/

is very common with 28 projects doing it, while 13 projects maintain images for multiple versions and flavours.

> The reasons for maintaining several Dockerfiles in official projects are to support multiple: versions, base-images, flavours and platforms. The most common reason combinations being: version/base-image and version/flavour.

V. RQ2: Do duplicates arise in Dockerfiles families and why?

To answer RQ2, we report statistics about extracted duplicates (Section V-A) and confront them to our survey results. Finally, we manually examine a random subset of duplicates and analyse the reasons behind their existence (Section V-B).

A. *Duplicates and Co-evolution Statistics*

In this section, we present some statistics regarding duplicates we have identified with our detection tool. We also take a closer look at the maintenance surrounding these duplicates through a co-evolution analysis.

1) Methodology: As we said in Section III-C, our tool identified 877 duplicates in our repositories. Based on this, we compute several statistics about the characteristics of these duplicates. We also take a closer look at the Dockerfile DSL instructions composing duplicates, especially, which instructions are most commonly concerned by duplicates.

We then evaluate if these duplicates have an impact on DF maintenance by analysing if it's common practice for DFM to perform the same modification on several Dockerfiles of the same DF as we have discussed in Section I. To that extent, we automatically look at every commit of our repositories using the Diggit tool[13] seeking for commits that had two Dockerfiles or more being edited with the exact same modifications (a modification being a sequence of removed code and a sequence of added code, as computed by *diff*).

2) Duplicates: Figure 7 (left) depicts a boxplot for percentage of duplicate instructions per project. We notice that 75% of projects have nearly half of their instructions that are duplicated. Half the projects have more than 83% of duplicate instructions. All of this indicates how frequent duplicates are in DF. When asking DFM of our repositories if they have faced duplicates in the past: 68% (17 out of 25) said that they did, while 20% said they've never faced duplicates and 12% didn't know. Figure 7 (right) presents a boxplot for the number of instructions for each identified duplicate. We notice that duplicates can be quite small, with half of them having less than 7 instructions. However, we also note that 25% of duplicates we have identified have more than 22 instructions. Finally, we remind that we chose to split *RUN* instructions as described in Section III-C, which could artificially increase the number of instructions in blocs of duplicates. Also, when we ask all DFM: *What should be the minimal size for a duplicate bloc of instructions to be detected?*, the majority of DFM (15 out of 25) stated it should be between 1 and 5 instructions which indicates that even duplicates composed of a single instruction shouldn't be ignored.

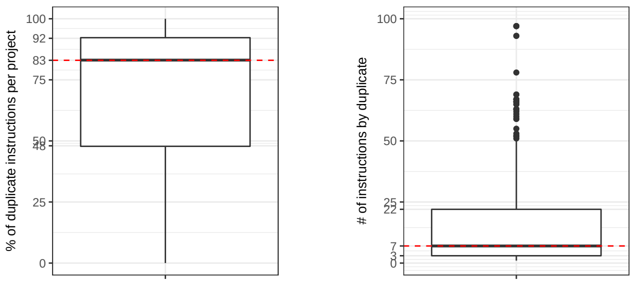


Fig. 7. Left plot: Boxplot for the percentage of duplicate instructions per project. Right plot: Boxplot for the number of instructions by duplicate.

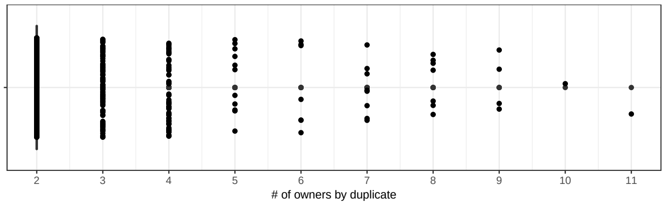


Fig. 8. Stripplot of the number of owners of every duplicate in our corpus.

Figure 8 is a stripplot depicting the number of owners (i.e. the number of files containing the duplicate) of every duplicate in our repositories. We notice that the large majority of duplicates has around 2 and 4 owners. While duplicates having from 5 to 11 owners are less common. When we ask DFM for the minimal number of files sharing a duplicate (owners) in order for that duplicate to be detected, 13 out of 25 DFM stated that it should be 2. These responses confirm that the minimal thresholds we had set previously in Section III-C were actually corresponding to the DFM needs.

Also, if we look at the ratio of duplicated instructions over the total number of instructions for every instruction provided by the Dockerfile DSL. We find that the high majority of instructions available in our corpus are duplicated (32,269 out of 37,319). The *RUN* instructions being the most frequent instructions in our corpus by far (26,053 instructions) where 86% of them are duplicated. The second most frequent instructions are *ENV* instructions (3,762 instructions) with more than 78% of them being duplicated. Followed by *FROM* instructions (1,372 instructions) with 79% of them being duplicated.

Finally, while our tool identifies 877 duplicates, 64% of all DFM from our survey (16 out of 25) said that they don't need a tool to detect duplicates. They state that duplicates are easy to find and that they don't want to use another tool. Therefore, the need for a detection tools for Dockerfile duplicates isn't as important as it's the case with duplicates in programming languages [3]–[5], however, 9 maintainers are nonetheless asking for such tools to help them ease the

[13] https://github.com/jrfaller/diggit

maintenance process.

3) Co-evolution: Figure 9 depicts a boxplot for the percentage of co-evolving commits per project in our corpus. We can see that 50% of all projects have 14% of all their commits propagating the same edit across several Dockerfiles. This number can go up to more than 29% for 25% of projects in our corpus.

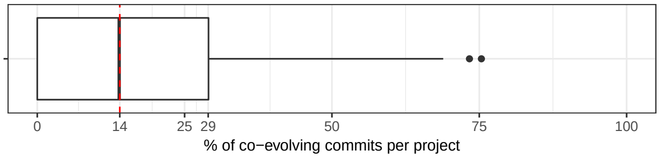


Fig. 9. Boxplot for the percentage of co-evolving commits per project.

Also, 94% of DFM who encountered duplicates (16 out of 17) stated that they had performed identical changes across multiple Dockerfiles in a single commit in the past. This confirms that keeping consistency across duplicates is a classical task performed by maintainers. Half of them (8 out of 16) qualified these consistency updates as being annoying. 4 out of 16 DFM felt that these consistency updates can be error-prone. However, others (6 out of 16) qualified them as being easy to perform since they can be automated through templates and other techniques as we'll see in Section VI.

Duplicates in DF are frequent. While they are usually small and span across only few Dockerfiles, they can also sometimes be large and span across many Dockerfiles. DFM of our corpus frequently have to propagate identical changes to multiple Dockerfiles. They also have a mixed opinion about these changes. DFM not using any tool find them error-prone and annoying, while DFM using dedicated tools find them easy to perform.

B. Reasons for Duplicates

In this section, we aim at understanding the underlying reasons behind duplicates we have identified earlier.

1) Methodology: For this study, we take a random sample of 50 duplicates from our duplicates corpus. We limited the size of this random selection to only 50 duplicates, because analysing a duplicate takes a long time. Three of the authors, external to the DFM of our corpus, review each duplicate and tag them with what they think is their corresponding reason (see Section VII for potential threats). Note that a single duplicate can have multiple underlying reasons.

2) Results: In the remainder of this section, we describe the main reasons behind duplicates.

a) Software installation and configuration: The installation process of a software can be identical across multiple images, leading to duplicate instructions. The process can be composed of different steps going from downloading the software and installing it, to checking its signature, etc.. Once a software is installed, the next step in this process is to configure the software before running it.

Listing 4. Duplicate due to identical software installation process

```
1  ENV XWIKI_VERSION=8.4.5
2  ENV XWIKI_URL_PREFIX "http://maven.xwiki.org/.../
       ${XWIKI_VERSION}"
3  ENV XWIKI_DOWNLOAD_SHA256 52ed122c44984748a729a78
        \
4  4c94cb70ccf0d2fa34c2340d0fd45c75deb3b0bc9
5  RUN rm -rf /usr/local/tomcat/webapps/* && \
6  mkdir -p /usr/local/tomcat/temp && \
7  mkdir -p /usr/local/xwiki/data && \
8  curl -fSL "${XWIKI_URL_PREFIX}/xwiki-ent-web-${
       XWIKI_VERSION}.war" -o xwiki.war \&& \
9  echo "$XWIKI_DOWNLOAD_SHA256 xwiki.war" |
       sha256sum -c - && \
10 unzip -d /usr/local/tomcat/webapps/ROOT xwiki.war
        && \
11 rm -f xwiki.war
```

Listing 4 is an extract of a whole duplicate (11 lines) that we manually analysed This duplicate involves two files from the *XWiki* project. It first starts by defining some environment variables (lines 1 to 4), then cleans a folder, and creates other ones that are mandatory for running XWiki (5 to 7). It then downloads the software and puts it in the right folder, validates it, unzip it and deletes the downloaded file. We encountered this reason for 50% of duplicates we analysed.

b) Dependency management: Dependencies also follow an installation process. Before installing dependencies, package managers need to be configured. After having installed the dependencies, it is sometimes necessary to configure it. Finally, the list of dependencies that needs to be installed may vary across images, but most dependencies are identical.

Listing 5. Duplicate due to package manager configuration

```
RUN apt-key adv --keyserver pgp.mit.edu \
--recv-keys 1614552
    E5765227AEC39EFCFA7E00EF33A8F2399
RUN echo "deb http://download.rethinkdb.com/apt
    xenial main" \
> /etc/apt/sources.list.d/rethinkdb.list
```

Listing 5 is an extract of a duplicate caused by the package manager's configuration to download the dependency. It involves three Dockerfiles from the rethinkdb project [14].

Listing 6. Duplicate due to dependency manager installation

```
1  RUN apt-get update
2  RUN apt-get install -y mysql-client-core-5.7
       openjdk-8-jre-headless postgresql-client
       unzip curl zip
3  RUN rm -rf /var/lib/apt/lists/*
```

Listing 6 shows a duplicate due to the installation of identical dependencies. In this extract, the package manager starts by updating its list of packages (line 1). Then, it downloads a bunch of dependencies, and finally, it cleans the package list it previously downloaded in order to reduce the final image size. It involves 2 files of the the bonita project [15]. It is common to see the underlying shell commands combined in a single RUN instruction as we explained in Section II. We encountered this reason for 40% of duplicates we analysed.

[14] https://github.com/rethinkdb/rethinkdb-Dockerfiles

[15] https://github.com/Bonitasoft-Community/docker_bonita

c) Runtime configuration: These duplicates arise because developers set-up a same way of running the container in several images. Indeed, while writing the Dockerfile, it's possible to configure some parameters for the container runtime. For instance, the instruction *ENTRYPOINT* lets developers configure a shell command that will be run inside the image when instantiated as a container. For example, a bash image when started, will directly run the bash binary. There are also instructions such as *VOLUME* for specifying how to mount a folder from the user computer into the container, instructions for specifying the working directory, etc.

Listing 7 shows a duplicate due to runtime configuration. In this extract, the Dockerfile specifies where a volume should be mounted (line 1), what command will be used as an entrypoint (line 4) and what command should be used (line 5). It involves 2 files from the *spiped*[16] project. We encountered this reason in 26% of duplicates we analysed.

Listing 7. Duplicate due to runtime configuration

```
1 VOLUME /spiped
2 WORKDIR /spiped
3 COPY *.sh /usr/local/bin/
4 ENTRYPOINT ["docker-entrypoint.sh"]
5 CMD ["spiped"]
```

Duplicate instructions are due to: software installation and configuration, dependency management and runtime configuration.

VI. RQ3: What are the tools used by experts to manage Dockerfiles with their pros and cons?

To answer RQ3, we start by explaining the methodology we follow to identify the tools used by experts, we present the three categories of tools we discover, and then we discuss their advantages and limits.

A. Methodology

To answer our third and final research question, we manually analyse each repository in our corpus to determine if what we call a *Dockerfile management tool* has been used. We define as *Dockerfile management tool* any tool that helps the maintenance of Dockerfiles. To that extent, we manually look at all 99 projects in our corpus looking for scripts or binaries that could be used as tools to manage Dockerfiles. Then, we take a closer look at each of these scripts trying to understand the implemented Dockerfile management strategy. Next, we cluster theses scripts and binaries and identify 3 main Dockerfile management tool categories (see Section VII for potential threats). We perform a thorough analysis to describe each one of these categories (Section VI-B. We also report our survey's answers relative to the use of tools. Finally, we discuss the tools with regard to their capabilities to handle change propagation, highlighting their advantages and their limitations (Section VI-C).

B. Dockerfile Management Tools

We find that 66 projects in our repositories use tools to manage their DF (66% of our corpus) . We notice that there is no off-the-shelf tool for this purpose: all developers maintain their own tool. However, all management tools have an *update script file* (located in the root of the repository) that is usually named `update.sh`, which is responsible for updating the whole DF. The update script needs some input parameters to generate the Dockerfiles, such as the version number or the target base image. 75% of them receive the parameters from the command-line while 25% automatically fetch parameters from a website. Finally some of the latter projects even automate the execution of the `update.sh` script at regular intervals via a dedicated bot. We notice that 88% of these scripts are written in *Bash*, the remaining 12% are written in different languages such as *Go*, *PHP*, *Makefile*, *Python*, *Perl* and *Groovy*.

Further, when we ask DFM in our survey if they use tools to handle duplicates, 56% (14 out of 25) said they actually don't, because it is too much time consuming or difficult to implement. When we ask them: *Would you like to have a tool built to avoid duplicates in Dockerfiles?*, about 57% (8 out of 14) said they would like to have one, which confirms the usefulness for management tools. The remaining ones (6 out of 14) said no because they don't want to use another tool or don't see the need. However, among the remaining 44% (11 out of 25) who are using tools, 9 replied they were using the tools that we'll present in this section, and 2 replied that they were using git branches and multi-stage builds as management tools for their Dockerfiles. 7 out of the 11 DFM using tools said that they were satisfied with their tools, remaining 4 weren't satisfied with their tool. However, by analyzing the answers, we realized that all maintainers multiple different tools depending on the project. Therefore, we could not use the answers of DFM to directly pinpoint their favorite tools.

We now present the main categories of Dockerfiles management tools we encountered.

a) Template processor: They are the most frequent tools in our corpus (54%) and are used by 8 out of 11 maintainers in our survey. Template processors use a template, some input data, and a template engine. This kind of tools are also widely used to generate web pages [6], [7]. When invoked, the template engine injects the input data into the templates to generate the outputs.

Listing 8. An extract of the python Dockerfile template

```
1 FROM debian:%%PLACEHOLDER%%
2 ENV GPG_KEY %%PLACEHOLDER%%
3 ENV PYTHON_VERSION %%PLACEHOLDER%%
```

The most classical templates used in our corpus are just plain Dockerfiles containing several placeholders as done in Figure 8. This listing presents an extract of the template used to generate the Dockerfiles of the python project. We first notice that the base image is defined by a text replacement and a variable (line 1). The GPG key and python version that needs to be installed use the same features (text replacement

[16] https://github.com/TimWolla/docker-spiped

and a variable (lines 2 and 3)). Only a few projects use more advanced template technologies supporting advanced features such as inclusion of sub-templates, loops and conditional statements: only 2 projects implement sub-templates, and only one project supports conditional statements. For instance, the *XWiki* project uses the most advanced template language we observed. Listing 9 shows an extract of this project's template. Among the list of XWiki's dependencies, developers must install either *mysql* or *postgres* according to the desired image flavour In the example, we see that a conditional statement is used (coming from Groovy's templating language) to handle this case (lines 7 and 8). Regardless of the template language, projects using templates all perform the template rendering inside the `update.sh` script.

Listing 9. An extract for the XWiki Dockerfile template

```
1  RUN apt-get update && \
2   apt-get --no-install-recommends -y install \
3    curl \
4    libreoffice \
5    unzip \
6    procps \
7    <% if (db == 'mysql') print 'libmysql-java'
8    if (db == 'postgres') print 'libpostgresql-jdbc
        -java' %> && \
9   rm -rf /var/lib/apt/lists/*
```

b) Find and replace: Find and replace tools are also a fairly common Dockerfile management tools, and are used by 36% of the projects in our corpus and by 2 out of 11 maintainers in our survey. These tools proceed to update all Dockerfiles present in the repository by directly updating some of their content using the input data. Therefore, Dockerfiles present in the repository are overwritten by the updated ones.

Such Dockerfile management tools mainly use regular expressions or dedicated Unix tools such as *sed*, located directly in the `update.sh` script. For instance, Listing 10 depicts a real extract from Kibana project's update script. We see that the values at the right of *KIBANA_MAJOR*, *KIBANA_VERSION* and *KIBANA_SHA1* are replaced by the value contained in the variables passed as input parameters to the update script.

Listing 10. An extract of the Kibana update script

```
sed -ri '
  s/^(ENV KIBANA_MAJOR) .*/\1 '"$version"'/;
  s/^(ENV KIBANA_VERSION) .*/\1 '"$fullVersion"
     '/;
  s/^(ENV KIBANA_SHA1) .*/\1 '"$sha1"'/;
' "$version/Dockerfile"
```

c) Generator: Generators are the least frequent Dockerfile management tool, only used by 10% of the projects in our corpus. However, they are used by 8 out of 11 DFM in our survey. They consist of a single update script that generates all Dockerfiles with their content using a shell language (bash for 4 out of 5 projects) or a general-purpose language (perl for one project). These tools leverage on features offered by their host language and therefore provide many features (variables, loops, conditional evaluation, functions, ...). Listing 11 is an extract from the OpenJDK project's update script (written in bash). In this example, we see that a loop is used to generate the Dockerfiles for all versions of OpenJDK. Additionally, a conditional evaluation is used to add some extra dependencies if they are available for the target base image. Tools using generators place all the generator's code directly the `update.sh` script.

Listing 11. An extract for the OpenJDK Dockerfile generator

```
for version in "${versions[@]}"; do
 ...
 if [ "$addSuite" ]; then
  cat >> "$version/Dockerfile" <<-EOD
   RUN echo 'deb http://deb.debian.org/debian
        $addSuite main' > /etc/apt/sources.list.d
        /$addSuite.list
  EOD
 fi
done
```

> Many projects use tools to handle duplicates. They fall into three categories: find and replace, template processors and generators. Several DFM stated that such tools can be too much time consuming or difficult to implement, and thus are not using them. However DFM using tools are mostly satisfied with them.

C. Discussion

In this section, we review how each kind of Dockerfile management tool enables developers to handle duplicate code. In order to discuss the pros and cons of each tool, we start by taking a closer look at why DF co-evolve and therefore what types of changes DFM are propagated on Dockerfiles. To do so, we randomly select a sample of 50 commits having several Dockerfiles co-evolving from all co-evolving commits previously extracted in Section V-A3. We then manually look at each commit and determine the reason behind each co-evolution. As a result, we find two main types of changes: version update (28 out of 50) that are similar to the change shown in Listing 2 and other changes (22 out of 50), such as bug-fixes or refactorings, similar to the change shown in Listing 1. Version updates are predictable changes regularly performed at the same location, while the other changes are arbitrary and can span across multiple lines. We use these two categories of propagated changes to discuss the pros and cons of each category of tools in the remainder of this section.

a) Find and replace: Pros. Find and replace tools handle very well version updates propagation. Since the location of changes is known in advance, it's easy to build a regular expression or a sed command that automates them. In addition, it's very easy to set-up a find and replace tool, since it only requires to write a script that does nothing more than applying the sed command or the regular expression when called.

Cons. On the other hand, find and replace tools are not adapted to other changes than version updates. Indeed, these changes are usually only applied once on the code base, therefore there is no use for defining and storing a regular expression or a sed command to perform it. In case of such changes, developers have to find all Dockerfiles containing the duplicate sequence of instructions and apply the fix manually.

b) Template processor: Pros. Template processors are capable of handling the two change propagation scenarios. For both scenarios, it is sufficient to apply the change on the templates containing the concerned sequence of instructions, and re-generate the Dockerfiles. We note that projects using template processors and maintaining Dockerfiles for only one reason, usually write only one template, thus eliminating all possibles duplicates.

Cons. Projects using template processors and maintaining a DF for more than one reason write multiple templates (only four projects out of 26 in this case managed to write only one template). When we run the detection tool we used previously in Section III-C on the templates of projects using multiple templates (23 projects out of 35 projects using templates), we still find 95 duplicates in the templates. Figure 10 presents statistics about these projects. The left figure is a boxplot showing the percentage of duplicate instructions across Dockerfiles of projects using templates. The right figure shows a boxplot for the percentage of duplicates reduction in projects using templates. We can see that the use of templates helps reducing the number of duplicates that a DFM has to manage with a median reduction of 31%. However, duplicates aren't fully eliminated even when using templates, therefore DFM still have to propagate some changes manually.

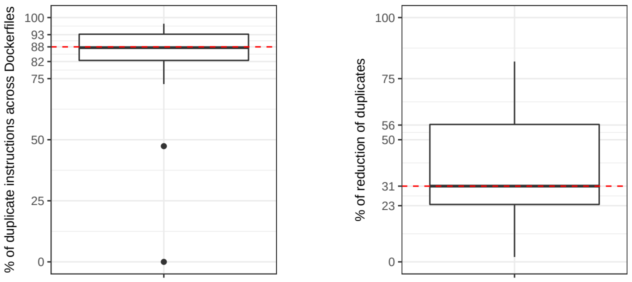


Fig. 10. Left plot: Boxplot for the percentage of duplicate instructions in Dockerfiles of a project using Templates. Right plot: Boxplot for the percentage of duplicates reduction in projects using templates.

c) Generator: Pros. Similarly to the template tools, generator tools are capable of handling both change propagation scenarios. Indeed, the five projects using these tools have a single file in their repository that generates all Dockerfiles, thus eliminating all duplicates. By looking at the source code of their generator, we found out that the key features were: text replacement, loops and conditional evaluation (as we can see in Listing 11). To propagate a change, the DFM have to locate the concerned location in the generator's code, perform the change, and regenerate all Dockerfiles.

Cons. The only downside we found with these tools is that the content of the generator (usually a shell script) is very cumbersome. Indeed, while DFM using templates write and read code that is very similar to a Dockerfile, DFM using generators write Dockerfiles with a totally different language. Additionally, it is far from obvious to understand the content of Dockerfiles that will be generated from it.

Find and replace tools handle very well the propagation of version updates but aren't adapted for other changes. Template tools are capable of handling the two change propagation scenarios but still don't fully eliminate duplicates. Generators are also capable of handling the two change propagation scenarios, they do fully eliminate duplicates but are much less readable than templates.

VII. Threats to validity

We discuss here the threats to validity of our study following the guidelines provided by Wohlin et al. [8].

Internal validity: In Section III-A, we look for Dockerfiles only based on their filename while supposing that the name is `Dockerfile`. We also assumed in Section IV-1 that the reason behind the existence of a Dockerfile is encoded in its path. Of course, it is possible that some projects use alternative naming and location schemes as this would bias our results. To ensure the validity of our hypothesis we manually inspect a random subset of 20% of projects and don't find any counter-example. In Section III-B, the survey gathered 25 responses from DFM. While this number isn't large, the fact that they are experienced DFM of official Docker repositories means that their answers can be considered of high-quality. Also, since we are aiming at only official repositories DFM, the answers we gathered could be biased towards a more advanced category of Dockerfile maintainers. In Section V-B, while the sample size is consequent, it isn't large enough to be representative of the whole duplicates corpus. Further, duplicates' reasons tagging is done by the paper's authors who aren't experts of the projects containing the duplicates. Moreover, the process of tagging a duplicate is very subjective which could bias the tagging results. Finally, while the three authors tried to mitigate the misidentification threat by concerting each other, a more formal approach should be taken in a future study replication work. In Section VI-A, Dockerfile management tools are identified by two authors of the paper. While strategies aren't that similar, meaning less subjective to identify, the two authors concerted each other to mitigate the misidentification threat. We attempt to provide all the necessary details to replicate our study and analysis. We also provide all data involved in our paper to enable replication and scrutiny of our results[17].

External validity: This study focuses only on mature and official projects, corresponding to the 99 projects in our repository. This is on purpose, as these projects are defined by Docker as implementing all best practices. Therefore, the projects may be more maintained and have larger Dockerfiles than more common projects meaning that the results we got may not be representative of the whole Docker ecosystem but to a rather more advanced category of projects.

[17] https://se.labri.fr/a/ICSME19-docker-oumaziz

VIII. Related work

In this section we discuss some research related to Docker and infrastructure as code. We also discuss different techniques used to detect duplicates in source code and other artefacts.

Cito et al. [9] conduct an empirical study on over 70,000 Dockerfiles assessing their quality and identify types of changes between consecutive versions of Dockerfiles. They show that common Dockerfiles are not changed often, while they indicate that popular Dockerfiles (like the ones in our corpus) happen to be more actively updated. They also indicate that the RUN instruction is the most commonly used instruction in Dockerfiles which can be linked to our results in Section V-A where we indicate that RUN instructions are the most frequently duplicated. Zerouali et al. [10] perform an empirical analysis on 7,380 official and community Docker images that are based on the Debian Linux distribution. They look at the relation between outdated containers and vulnerable and buggy packages. They show that no release is devoid of vulnerabilities and that even if they wanted, maintainers couldn't avoid them even with the most recent packages. Jiang and Adams [11] through an empirical study on over 256 OpenStack projects analysing infrastructure specification files show that these files are actually large and are frequently updated (28% changed monthly) which could lead to potential bugs. They also show that infrastructure files are tightly coupled to test files which leads to testers frequently changing infrastructure specifications when modifying their tests.

Sharma et al. [12] propose 11 implementation and 13 design code smells for configuration (infrastructure) code. They analyse 4,621 puppet projects for their identified code smells and show that up to 25% of their analysed repositories have code that is duplicated. They also analyse the correlation between their code smells, and show that design smells happen to co-occur more frequently which indicates the importance of design choices taken by developers. Hummer et al. [13] propose a framework for testing Infrastructure as code automations for idempotence by following a model-based testing approach. They evaluate their tool on over 300 Chef scripts and determine that almost the third of them where actually idempotent. Their tool even let them to actually detect and report a bug in Chef implementation itself.

Existing research on duplicates detection can be divided in two categories as follow:

(a) *in source code:* CCFinder [3] proposes a token-based detection technique to identify duplicate code. White et al. [14] used a learning-based detection technique which relies on deep learning to detect code duplicates. They found that their approach detected duplicates that weren't or where sub-optimally detected by traditional techniques. Ducasse et al. [15] developed a language independent technique for detecting duplicated code without using any parser. Their technique relied on line-based string matching. There are also other duplicate detection techniques such as AST-based techniques [16]–[19] and Graph-based techniques [20], [21]. Rattan et al. [22] did a systematic review of all software clone detection tools, regrouping other duplicates detection techniques we didn't present here.

(b) *in other artefacts:* Liu et al. [23] propose a suffix-tree based approach to detect duplicates in UML sequence diagram. Störrle [24] also explores duplicates in UML and implement algorithms and heuristics for detecting duplicates with the MQlone tool. Juergens, Domann et al. [25], [26] apply code detection techniques to 28 requirement specifications and discuss the nature and consequences of such duplicates Deissenboeck et al. [27] apply an automatic clone detection technique for large control systems models. McIntosh et al. [28] analysed 3,872 build systems looking for clones, they identified what are the underlying reasons behind their existence and which recent build technologies tend to be more prone to cloning. Oumaziz et al. [29] performed an empirical study on documentation reuse. They applied a clustering technique (formal concept analysis) to detect duplicates in documentation. And showed that existing documentation tools lack reuse mechanisms for situations such as code delegation.

IX. Conclusion

Docker is becoming a popular tool used by developers and end-users to deploy and run software applications. Dockerfiles are now found alongside projects' source code. Several projects are even starting to maintain families of Dockerfiles. In some situations, Dockerfiles family maintainers have to propagate a change to several, if not all, Dockerfiles of the family. This need to propagate changes is usually due to the presence of duplicates between several family's Dockerfiles.

In our study, we answer the question of duplicates handling in Dockerfiles families by providing a grounded study based on the analysis of the *Official Docker Projects*. We show that official Docker projects frequently maintain families of Dockerfiles and find the underlying reasons: supporting multiple versions/base-images and versions/flavours.

We then show that duplicates in Dockerfiles are abundant, and find the underlying reasons behind them: Software installation and configuration, dependency management and runtime configuration. We also perform a survey on DFM of official projects and find that they are aware of their existence and are frequently facing them. However, DFM have a mixed opinion regarding them. While DFM not using tools for handling duplicates state that their handling may be error-prone, DFM using tools state that they are easy to maintain.

We also find that some DFM handle duplicates by using ad-hoc tools: template processors, code generators, find and replace executors. Then, we show that projects using template processors and code generators manage to reduce the amount of duplicates with a median at 30% up to 100% for generators.

As a future work, we plan to extend our study towards projects maintained by the broad community of maintainers in order to identify broader needs. We also plan to work on a management tool that could be easy to use out of the box by the community of DFM in order to ease the existing initial burden of migrating to management tools.

REFERENCES

[1] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–9.

[2] A. Lex, N. Gehlenborg, H. Strobelt, R. Vuillemot, and H. Pfister, "Upset: visualization of intersecting sets," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 1983–1992, 2014.

[3] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 1998, pp. 368–377.

[5] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on software engineering*, vol. 33, no. 9, 2007.

[6] T. J. Parr, "Enforcing strict model-view separation in template engines," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04. New York, NY, USA: ACM, 2004, pp. 224–233. [Online]. Available: http://doi.acm.org/10.1145/988672.988703

[7] M. Tatsubori and T. Suzumura, "Html templates that fly: A template engine approach to automated offloading from server to client," in *Proceedings of the 18th International Conference on World Wide Web*, ser. WWW '09. New York, NY, USA: ACM, 2009, pp. 951–960. [Online]. Available: http://doi.acm.org/10.1145/1526709.1526837

[8] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[9] 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 the 14th International Conference on Mining Software Repositories*, ser. MSR '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 323–333. [Online]. Available: https://doi.org/10.1109/MSR.2017.67

[10] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "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, 2019, pp. 491–501.

[11] Y. Jiang and B. Adams, "Co-evolution of infrastructure and source code: An empirical study," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 45–55.

[12] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 2016, pp. 189–200.

[13] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam, "Testing idempotence for infrastructure as code," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2013, pp. 368–388.

[14] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 87–98.

[15] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE, 1999, pp. 109–118.

[16] W. Yang, "Identifying syntactic differences between two programs," *Software: Practice and Experience*, vol. 21, no. 7, pp. 739–755, 1991.

[17] R. Falke, P. Frenzel, and R. Koschke, "Empirical evaluation of clone detection using syntax suffix trees," *Empirical Software Engineering*, vol. 13, no. 6, pp. 601–643, 2008.

[18] R. Tairas and J. Gray, "Phoenix-based clone detection using suffix trees," in *Proceedings of the 44th annual Southeast regional conference*. ACM, 2006, pp. 679–684.

[19] D. Gitchell and N. Tran, "Sim: a utility for detecting similarity in computer programs," in *ACM SIGCSE Bulletin*, vol. 31, no. 1. ACM, 1999, pp. 266–270.

[20] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 321–330.

[21] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *International Static Analysis Symposium*. Springer, 2001, pp. 40–56.

[22] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[23] H. Liu, Z. Ma, L. Zhang, and W. Shao, "Detecting duplications in sequence diagrams based on suffix trees," in *Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific*. IEEE, 2006, pp. 269–276.

[24] H. Störrle, "Towards clone detection in uml domain models," *Software & Systems Modeling*, vol. 12, no. 2, pp. 307–329, 2013.

[25] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit, "Can clone detection support quality assessments of requirements specifications?" in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 2010, pp. 79–88.

[26] C. Domann, E. Juergens, and J. Streit, "The curse of copy&paste cloning in requirements specifications," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 2009, pp. 443–446.

[27] F. Deissenboeck, B. Hummel, E. Jürgens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert, "Clone detection in automotive model-based development," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 603–612.

[28] S. McIntosh, M. Poehlmann, E. Juergens, A. Mockus, B. Adams, A. E. Hassan, B. Haupt, and C. Wagner, "Collecting and leveraging a benchmark of build system clones to aid in quality assessments," in *Companion proceedings of the 36th international conference on software engineering*. ACM, 2014, pp. 145–154.

[29] M. A. Oumaziz, A. Charpentier, J.-R. Falleri, and X. Blanc, "Documentation reuse: Hot or not? an empirical study," in *International Conference on Software Reuse*. Springer, 2017, pp. 12–27.