Refactorings and Technical Debt in Docker Projects: A Replication Study

Billy Bouchard, *billy.bouchard@polymtl.ca*
Quentin Guidee, *quentin.guidee@polymtl.ca*

***Abstract*—This document is intended to be a replication study of the work of [1] on the refactorings and technical debts in Docker projects. Our team tried to do a second pass on around 10% of the main study results. The goal is to observe how different/similar our conclusions were. This double check led us to observe that the intial conclusion of [1] were mainly focused on performances. We were able to add a total of 3 new refactoring types. We also noticed that the results of the main study are very subjective and debatable. They should have defined the refactoring types more precisely to avoid confusion.**

I. Introduction

Distributed systems have many uses, but also many flaws, one of which being the lack of standard between the systems that you deploy on. Docker [2] and its containers try to help in that regards, focusing on giving a simple standard interface from which the machine used and its system matters less.

These tools let the developer script the way a systems is supposed to work, independently from the operating system (OS). Docker images can then be reused and redeployed elsewhere, so that building the same project on another machine is only a matter of minutes. By their own nature, containers are most often used in complicated systems and are, by that very same definition, usually hard to maintain. Well maintained Docker projects can save developers a lot of headaches and time, and even give them powerful tools.

However, poorly maintained ones can be even more frustrating than directly developing on the OS. The research from [1] tried to categorize the different refactoring practices in Docker systems, and find those that are used the most. Few studies before the one being replicated have studied the maintenance cycle of Docker projects. Thus, the importance of this study and of the work of categorisation it has done.

This replication will challenge the methodology process, and the results [3] obtained by [1]. The main focus of this replication is not to add more commits or projects to the pool, but to look at the already analyzed data and verify the results and conclusions.

II. Methodology

The research [1] investigated a total of 68 open-source projects with over 19 MLOC. The selection of these projects was done with the help of the GitHub BigQuery archive, that contains a total of 2000+ Docker projects. Their selection process skimed the projects that were forked from others, and the non-existing ones. Looking into keywords, any commits containing the word `REFACTOR` has been kept. This resulted in 4,469 commits with a maximum of 73 commits per repository, finally keeping only 68 projects and 193 commits that were mostly written in Java. The commits ranged from 1 to 12 per project, with a total of 611 files changed.

The replication was performed on around 10% of the data from the first study (62 file changes at most). The process for the selection of this subset was initially random based. However, further looking into the data, we decided to keep the same ratios of Docker and docker-compose files. Finally, taking the whole list of commits from the replication package, we took all commits which are multiple of 8. This allowed to spread all the analyzed data equally across projects and changes, leading to a fair number of commits from each projects in our subset of data. A manual check was then done on all selected changes, where the study conclusion was looked at to be either confirmed or infirmed.

> **RQ1:** Are the main article conclusions accurate?

From the manual check, 4 possible outcomes were looked at:

- The initial conclusion is correct.
- The change is not a refactor.
- The change is a refactor but of a different type.
- Something that is assumed not to be a refactor is deduced to be one.

Once all selected commits were verified, we were able to gather both the precision and the recall metrics. Not only did we note all the changes we would make to the results, but also the changes we did not retain after debate. We did this because we wanted to know how subjective and debatable these changes are.

After a first pass, we noticed a tendancy for the considered refactors to be oriented toward the performances, more than readability or understandability. A second check was then specifically done on the refactored sub-data to see if the refactoring types could be expanded. The goal of this second phase was to see if the initial conclusion of the study really considered all the possible refactors of a Docker or docker-compose file.

> **RQ2:** Can the main article refactor categories and types be expanded?

At the end of the second pass, we gathered groups of commits that we thought might bring new categories to the article. Our team added the categories that all were in favor of.

III. Results

Our results are available on GitHub[1]. Two main files are made available: `docker-file.md` and `docker-compose.md`. These files contain our results for the Dockerfile and docker-compose files respectively. Although we separated the tables between the team members, a common discussion was carried out, as described in the Section II. A checkmark in the *notes* column means that the conclusion of the initial study was confirmed by minimum one of our team member. Otherwise, this is either a mistake in the data of the original report, or a comment that we had to make to nuance their results.

The first major thing we noticed is the very important subjectivity of the results. This is probably due to a lack of precise definition of refactoring types by the original study. Not only are there debates within our team about some refactorings, but this problem can also be found in the original report between authors. For example, the *Update RUN Instruction* type is subjective. As stated in our replication data, there is a commit that the authors considered as an update to the run command, where the operator `&&` was moved from the end of a line to the beginning of the next line. That being said, another commit adding a space to the command is not considered as a refactoring. According to us, this is a major problem in the definitions of refactoring types, that can lead to inconsistencies in the results.

In the continuity of this first problem, we also note inconsistencies in the descriptions of the commits that the authors have made. Some descriptions describe all changes made in a commit, whereas others only describe the changes that are considered as refactoring. Although this issue may not be reflected in the final results, it is important to take this into account in case someone wants to reuse these results in another study.

Compiling all our observations, we were able to calculate both the recall (calculated as the amount of commit that rightfully were refactors) and the precision (calculated the amount of correct assumptions by the authors). The recall was of $\frac{70}{72}$ or 97 % which is really good considering the subjective quality of the work being done here. The precision, however, was calculated as $\frac{43}{72}$ or 59.7 %.

As for the RQ2, we notice 3 different changes that weren't part of the intial study, but that we think should be added. First, removing, adding or rewritting comments. As much as this type of change does not affect performances, we think it does have an impact on the readability, and therefore should be considered. Second, resectionning or transfering commands between files. The authors seems to have focus on each files individually during their study, which is normal as there is a huge amount of commits. However, when looking at a commit as a whole, we see sometimes some part of the code being moved from Dockerfiles to docker-compose files. The opposite also happens. Third, some variables were extracted and not removed. Some of the removal inside Dockerfiles were actually variables being moved to their own Dockerfile. This was considered by the authors as a removed variable, but we think it should be a different type called "extracting".

[1]https://github.com/34yu34/log6306-Replication

IV. Validity

This replication study shares validity threats with the study from [1] which it replicates. More specifically, it shares all the same issues with data gathering than the first study, since the data taken were exactly the same.

However, some other concerns might arise from the subset selection process, which might have not been representative of the total data. To minimize the impact, we tried to take commits from all the projects. Moreover, we sampled in the same ratios as the initial study the amount of Dockerfile versus the amount of docker-compose files.

Other threats to validity come from the subjective nature of the study. To minimize the impact on that end, we decided that unless all team members agreed on a difference, the study initial results would be kept. Also, every type added went through a discussion from the whole team where every member had to agree in order for that category to be added.

V. Conclusion

This study helped us dwelve deeper into the different refactors that happen around Docker projects. We were able to see that manual checking can lead to a very variable array of solutions. We also observed that new potential categories could be added to the projects to add more variety to the refactorings proposed by the main article authors.

As [1] says, the next step of development should aim at creating new automated approaches, to quickly find refactoring opportunities and propose them to code authors.

References

[1] E. Ksontini, M. Kessentini, T. d. N. Ferreira, and F. Hassan, "Refactorings and Technical Debt in Docker Projects: An Empirical Study," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2021, pp. 781–791, iSSN: 2643-1572.

[2] "Docker: Accelerated, Containerized Application Development," May 2022. [Online]. Available: https://www.docker.com/

[3] "ASE'21." [Online]. Available: https://sites.google.com/view/ase21-docker-refactorings