2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678585

Refactorings and Technical Debt in Docker Projects: An Empirical Study

Emna Ksontini, Marouane Kessentini, Thiago do N. Ferreira and Foyzul Hassan
University of Michigan-Dearborn, Dearborn, MI, USA
{emna, marouane, thiagod, Foyzul}@umich.edu,

***Abstract*—Software containers, such as Docker, are recently considered as the mainstream technology of providing reusable software artifacts. Developers can easily build and deploy their applications based on the large number of reusable Docker images that are publicly available. Thus, a current popular trend in industry is to move towards the containerization of their applications. However, container-based projects compromise different components including the Docker and Docker-compose files, and several other dependencies to the source code combining different containers and facilitating the interactions with them. Similar to any other complex systems, container-based projects are prone to various quality and technical debt issues related to different artifacts: Docker and Docker-compose files, and regular source code ones. Unfortunately, there is a gap of knowledge in how container-based projects actually evolve and are maintained.**

In this paper, we address the above gap by studying refactorings, i.e., structural changes while preserving the behavior, applied in open-source Docker projects, and the technical debt issues they alleviate. We analyzed 68 projects, consisting of 19,5 MLOC, along with 193 manually examined commits. The results indicate that developers refactor these Docker projects for a variety of reasons that are specific to the configuration, combination and execution of containers, leading to several new technical debt categories and refactoring types compared to existing refactoring domains. For instance, refactorings for reducing the image size of Dockerfiles, improving the extensibility of Docker-compose files, and regular source code refactorings are mainly associated with the evolution of Docker and Docker-compose files. We also introduced 24 new Docker-specific refactorings and technical debt categories, respectively, and defined different best practices. The implications of this study will assist practitioners, tool builders, and educators in improving the quality of Docker projects.

***Index Terms*—Docker, containers, refactoring, technical debt, maintenance**

I. Introduction

The containerization of software applications has recently becoming popular in software industry to improve the reusability, modularity, portability, security and costs of systems and their development [35], [7]. Among containerization frameworks, Docker is the main containerization framework in the open-source community [7] and industry as 79% of IT companies use it [27]. Indeed, Docker enables packaging an application with its dependencies and execution environment into a standardized, self-contained unit, which can be used for software development and to increase the portability of the applications [7]. The contents of a Docker container are defined in a Dockerfile. The Docker-compose is a tool for running multi-container applications on Docker and it is defined by a Compose file format to orchestrate their execution. Source code repositories of Docker projects contain Docker and Docker-compose files, as well as regular source code files written in a traditional programming language, such as Java, to implement the app hosting the containers and facilitate their execution and synchronization with other features.

Similar to any other complex systems, container-based projects are prone to various quality and technical debt issues related to different artifacts: Docker and Docker-compose files and regular source code ones. Unfortunately, there is a gap of knowledge in how container-based projects actually evolve and are maintained. Few recent studies focused mainly on the detection of the quality issues related to the Dockerfiles in terms of the violation of the basic shell scripts practices [15], [14]. However, there is no holistic understanding of the quality issues of Docker projects that could affect different artifacts beyond just the shell scripts in the Dockerfiles. Furthermore, the correction of these issues via refactorings, defined as changes to improve the structure while preserving the behavior, is still not yet explore in the literature unlike other refactoring domains [1]. As Docker projects become more complex and expensive to maintain [7], it is critical to understand the refactorings that developers would apply.

In this paper, we address the above gap by conducting an empirical study on refactorings, i.e., structural changes while preserving the behavior, applied in open-source Docker projects. The new knowledge out of the empirical study includes the discovery of (a) the types of technical debt addressed and whether they are specific to Docker projects, (b) the refactoring types that are common in the different artifacts of Docker projects, and (c) new generalizable Docker-specific refactorings and technical debt categories, if any.

Studying the refactoring types and technical debt categories that are typically found within Docker projects can lead to new automated Docker-specific refactoring techniques, quality issues detection tools for Docker and Docker-compose files and associated source code, and automated Docker-specific refactoring mining tools and techniques. The implications of the empirical study of this paper will help to (i) understand how and why quality issues and technical debts appear in Docker projects and how refactorings would address those issues, (ii) design new automated tools to integrate novel Docker-specific refactorings, (iii) provide guidelines for best practices, and anti-patterns/bad smells for practitioners in evolving effectively Docker projects, and (iv) assist educators in teaching quality issues related to Docker projects.

978-1-6654-0337-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ASE51524.2021.00074

Authorized licensed use limited to: Université de Montréal. Downloaded on March 19,2023 at 18:04:54 UTC from IEEE Xplore. Restrictions apply.

We analyzed 68 projects, consisting of 19,5 MLOC, along with 193 manually examined commits that include refactorings. The analyzed refactorings were labeled as being performed in Docker and Docker-compose files or regular code files, and related to the Docker-specific debt they alleviated. For the regular code refactorings, we used RefMiner [33] to detect the refactorings in Java files. With the identified refactorings in Docker projects, we developed a refactoring taxonomy. The results indicate that developers refactor these Docker projects for a variety of reasons that are specific to the configuration, combination, and execution of containers, by leading to several new technical debt categories and refactoring types compared to traditional source-focused refactoring domains. For instance, the reduction of the image size particularly involved refactorings of Dockerfiles, improving the extensibility is one of the main reasons for refactoring the Docker-compose, and regular source code refactorings are mainly associated with the evolution of Dockerfiles and Docker-compose. Our study indicates that (i) improving the build time, maintainability and reducing the image size are the main quality issues addressed in Dockerfiles, (ii) increasing the reusability, understandably and extensibility are driven most of the applied refactoring applied to Docker-compose files, and (iii) the evolution of the Docker and Docker-compose files resulted in various refactorings applied to the source code related to the Docker projects.

The main contributions of this paper can be summarized as follows:

- Based on the manual analysis of 68 Docker projects, we propose a rich taxonomy of generic and Docker-specific refactorings. Furthermore, this study provides to the community the first dataset on refactorings in Docker projects.
- We also introduce 24 new refactorings and 7 new technical debt categories specific to Docker projects.
- We propose recommendations, best practices, and anti-patterns for the evolution of Docker and Docker-compose files from our in-depth analysis of Docker projects.

Replication Package. All material and data used in our study are available in our replication package [4].

II. Background

A. Docker and Container-based Projects

Docker [10], is the most popular container virtualization technology [7], [27]. It aims on packaging application's code and dependencies into a light-weight, standalone and portable execution environment. Thus, deploying containerized applications is an agile process. Docker container images are built using `Dockerfile`, a document containing a sequence of instructions used for creating the computational environment, following the notion of Infrastructure-as-Code(IaC) [17].

Listing 1 illustrates an example of `Dockerfile`. It starts from a previously existing base image defined by the FROM instruction which acts as a starting point from which the Docker image will inherit infrastructure definitions. This parent image can be an official Docker image (e.g., alpine) or any other existing image (e.g., with a pre-installed software). In order to suit the application needs and to create the desired environment, Dockerfile offers a list of setup instructions which can be listed in Table I.

```
FROM node:argon
# Create app directory
WORKDIR /usr/src/app
# Install app dependencies
COPY package*.json /usr/src/app/
RUN npm install
# Bundle app source
COPY . /usr/src/app
# Expose the app to the outside world
EXPOSE 8080
CMD [ "npm", "start" ]
```

Listing 1: Dockerfile example.

TABLE I: Dockerfile setup instructions.

Instruction	Description
ENV	Setting the environment variables
ARG	Defining variables that can be set at build time
WORKDIR	Setting working directory for all subsequent instructions
COPY	Copying files from host to the Docker image
ADD	Similar to COPY instruction but supports two additional tricks. It supports the use of URL instead of a local file and can recognize the archive format and extract it directly into the destination
LABEL	Key value pairs, indicating image metadata
RUN	Executing any command
EXPOSE	Informs Docker that the container is exposing a particular port
CMD	Setting a command and/or parameters, that executes when the container is starting and which can be overwritten at build time
ENTRYPOINT	Setting executable that will always run when the container is initiated and cannot be overwritten.

In the Docker paradigm, each container captures one particular component of the software (e.g., database). Thus, when creating multi-component application using Docker, it is unavoidable to combine multiple software components (containers) into an intricate workflow. To tackle this challenge, containers need to be instantiated and properly integrated. Docker-compose [11], can be used to mitigate this challenge by providing a unified setup routine that deploys several containers using a YAML configuration file, as known as, `Docker-compose.yml` (or just Docker-compose).

```
version: "3.7"
    services:
        server:
            build: .
            ports:
                - 8080:4040
            environment:
                - DB_ADDRESS=database-mongo
                - DB_PORT=27017
                - PORT=4040
            depends_on:
                - database
        database:
            image: mongo:latest
            volumes:
                - mydata:/data/db
    volumes:
        mydata:
```

Listing 2: Docker-compose example.

An example of a Docker-compose file is available in Listing 2. The example shows that the Docker-compose

Authorized licensed use limited to: Université de Montréal. Downloaded on March 19,2023 at 18:04:54 UTC from IEEE Xplore. Restrictions apply.

file is composed of two components/containers (SERVER and DATABASE). The SERVER component is represented by a local image (built from a `Dockerfile`, for instance that one available in Listing 1) and the DATABASE component is created from the "mongo" image, hosted in DockerHub [12] (an online registry for Docker Images). Docker-compose file also provides a list of setup attributes which can be listed in Table II.

TABLE II: Docker-compose setup attributes.

Attribute	Description
BUILD	Setting path to the build context
IMAGE	Setting the image to start the container from
PORTS	Specify ports binding
ENVIRONMENT	Setting environment variables
DEPENDS_ON	Expressing dependency between services
VOLUMES	Setting volume bindings (host paths or named volumes)

Any typical Docker project includes the above files along with source code files written in a typical programming languages, such as Java, to host the containers and enable their executions and synchronization with other features of the app that may not be containerized.

B. Technical Debt and Refactoring

Software technical debt reflects the implied cost of additional rework caused by choosing an easy (limited) solution now instead of using a better approach that would take longer when designing and evolving software systems [6].

To deal with technical debt, refactorings are widely used practice [8]. Martin Fowler [13] defines refactoring as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior". This implies that refactoring is a method that reconfigures code structures, without altering its behavior, to improve code quality in terms of maintainability, extensibility, and reusability. Different refactoring types are defined in the literature including Move Method, Extract Method, Move Class, Move Attributes, etc. A full list of typical code refactoring types can be found in [22], [2], [3]. Recent empirical studies on refactoring show that these refactorings are widely used in open-source projects [31], [1], [28]. However, refactoring is still under-explored for Docker and containerization unlike other paradigms, such as object-oriented programming, web services, etc.

III. Methodology

Based mainly on manual analysis, we investigated the common refactorings in Docker projects. This study may present an empirical foundation for new refactoring types for the different artifacts of Docker projects to support practitioners in addressing Docker related technical debts.

A. Projects Selection

The proposed study includes a total of 68 open-source Docker projects as described in Table III. They differ significantly in their size and their popularity. These projects are comprising a total of 19+ MLOC with an average evolution history of 6+ years per project. We first selected Docker projects from the public GitHub archive on BigQuery [9], where our initial list included 2,342 open-source Docker projects. Then, we eliminated non-existing projects since BigQuery's last update was in 2019 and removed repositories forked from other repositories to avoid biasing our study, as large and popular projects are forked frequently.

We applied a selection criterion aiming to have at least one commit message mentioning the keywords REFACTOR and DOCKER, and at least one part of the project must include Docker. The number of commits having the required keywords was initially 4,469 commits with a maximum of 73 commits per repository. In our final selection, we focused mainly on a total of 68 Docker projects that were mostly written in Java for the code beyond the Docker and Docker-compose files, as it is a popular programming language to develop apps hosting containers [37]. However, we still also considered Docker projects with significant evolution written in other languages including C++. To support the manual analysis of the Java code, we used an assisting tool, RefMiner [33], but we note that most of the manual investigation efforts were mainly on the Docker and Docker-compose files to identify relevant Docker-specific refactorings beyond the traditional object-oriented refactoring.

The above selection mechanism yielded a total of 193 commits having the required keywords, ranging from 1 to 12 commit per project (column **KWS**) and a total of 611 Docker-specific file changes (column **DFC**). We manually examined the changes associated with these commits to find patches representing possible refactoring and addressed technical debts.

B. Commits Mining

To extracted commits that include refactorings from BigQuery achieve, we used the SQL query presented in Listing 3.

```
SELECT * FROM
"bigquery-public-data.github_repos.commits"
WHERE (message LIKE '%refactor%')
AND (message LIKE '%docker%')
```

Listing 3: SQL instructions for BigQuery.

The keywords were queried via the SQL like operator, where the % sign was used to represent zero, one, or multiple characters. The expression %REFACTOR% matches strings containing the word refactor (.e.g., refactoring, refactored) and the expression %DOCKER% matches strings containing the word docker (e.g., dockerfile, docker-compose).

C. Refactorings Identification

Since it is the first study about Docker-specific refactorings, it was necessary to manually inspect commits for refactoring identification. The keywords matching commits were chosen for manual examination to find patches in Dockerfile and Docker-compose files representing one or more possible refactoring, which required not only non-trivial efforts but also deep knowledge of the domain. Three of the authors have

Authorized licensed use limited to: Université de Montréal. Downloaded on March 19,2023 at 18:04:54 UTC from IEEE Xplore. Restrictions apply.

TABLE III: Studied projects.

Subject	KLOC	DFC	KWS
alebabai/linden-honey	4.8	8	8
all-of-us/workbench	350.7	2	1
amazeeio/lagoon	253.9	56	6
Artemkaaas/indy-sdk	352.5	6	4
aspuru-guzik-group/mission_control	87.3	2	1
Asqatasun/Contrast-Finder	23.8	16	1
bagage/cadastre-conflation	19.9	2	2
benbromhead/cassandra-operator	27.4	4	1
blobor/skipass.site	13.4	3	2
bookbrainz/bookbrainz-site	134.6	11	8
BSWANG/denverdino.github.io	618.4	1	1
BuiltonDev/pipeline	2800	11	2
byran/cyber-dojo-web	37.6	1	5
cdietrich/che	1200	4	1
cgi-eoss/ftep	1400	1	1
cloudfoundry-incubator/diego-release	1300	24	1
CogStack/CogStack-Pipeline	53.2	9	3
collinbarrett/FilterLists	1100	28	6
CrunchyData/crunchy-containers	39.5	4	2
CrunchyData/crunchy-postgresql-manager	447.4	22	3
CrunchyData/postgres-operator	105.7	3	2
cyber-dojo-languages/image_builder	2.2	13	3
cyber-dojo-retired/storer	7	10	6
cyber-dojo/commander	5.4	1	12
di-unipi-socc/DockerFinder	28.4	5	1
drasko/mainflux	915.6	2	4
duderoot/generator-jhipster	218.9	3	1
eclipse/repairnator	400.8	4	2
ethereum/hive	22.2	2	2
gchq/stroom	1300	3	1
geotrellis/geodocker-cluster	2.9	8	2
go-ggz/ggz	8	18	2
harvard-vpal/bridge-adaptivity	12.2	2	1
hexagonkt/hexagon	33.8	6	6
HumanExposure/factotum	515.6	23	7
InnovateUKGitHub/innovation-funding-service	816.8	8	1
instructure/straitjacket	183.4	4	1
ITISFoundation/osparc-lab	459.2	1	2
kr1sp1n/node-vault	6.3	2	1
kuzzleio/kuzzle	143.9	2	2
labsai/EDDI	103.3	8	1
lixiaocong/lxcCMS	7.1	4	2
lockss/laaws-metadataservice	4.5	1	1
luismayta/dotfiles	479.8	6	4
macarthur-lab/matchbox	85	1	1
mars-lan/datahub	79.3	17	5
Martin2112/trillian	119.2	3	1
MetaBarj0/carrier	13.1	6	1
mondediefr/mondedie-chat	10.9	6	6
muccg/rdrf	709	9	2
openpitrix/openpitrix	174	2	2
openzipkin/zipkin	111	20	3
ory-am/hydra	188.5	2	1
outlierbio/ob-pipelines	7.8	16	1
overture-stack/SONG	54.1	2	1
rackerlabs/blueflood	76.6	18	1
rancher/mesos-catalog	3.1	1	1
reportportal/service-api	55	10	9
RichardKnop/go-oauth2-server	9.8	7	1
robymes/OrdinglcDocker	42.7	24	10
scalableminds/webknossos	304.4	6	6
simonsdave/cloudfeaster	15	1	5
SKA-ScienceDataProcessor/integration-prototype	67.2	16	1
Soluto/tweek	74.2	8	2
stafli-org/stafli.stack.php	12.4	3	1
unbalancedparentheses/docker-erlang	5	69	1
vietj/vertx-pg-client	95.7	1	1
xhochy/arrow	1200	9	4
Total	19560.4	611	193

extensive expertise in refactoring, technical debt, and empirical software engineering. Another author is an expert in Docker and continuous integration including build repairs. Each of the four authors analyzed separately all the considered commits to identify the refactorings and later their rationale. They also discussed the identified refactoring at the end of the process (and not before to avoid any bias) to solidify the results especially when there are disagreements among the authors.

Cohen's Kappa coefficients [34] for refactoring identification and related commits, including Docker and Docker-compose files and regular source code files, were 0.92, 0.83, and 0.88, respectively, which indicates a high confidence of agreement. Since the authors may not be very knowledgeable about the code of the projects as they are not their original developers, they marked the refactorings and their associated commits only when they are very confident that the changes are actual refactorings. The authors also used commit messages and comments in the code whenever available to confirm their decisions, which is a common practice [19].

D. Refactoring Classification

After the refactoring identification phase and in order to understand the refactoring types performed in Docker projects, we analyzed the code changes within the selected list to determine the refactoring types and their rationale (e.g., technical debts), whether the refactoring was applied in a Dockerfile or a Docker-compose file or regular source code file (e.g., Java).

Discovered refactorings were then organized into a hierarchy based on the addressed technical debts. Hence, a few categories were grouped beneath distinctive parent categories within the hierarchy, i.e., change-sets containing a few interconnected refactorings were assembled into more common parent categories. A few of the refactorings were more disconnected, i.e., change-sets comprising one sort of refactoring and difficult to generalize.

Since the identified refactoring types of Dockerfile and Docker-compose may impact the regular source of the application hosting the containers, we have also manually investigated the selected commits in this study to look at the introduced code changes of Java/C++ files, within the same commit, whenever a Docker-specific refactoring is detected. To support the manual identification of refactoring, we used RefMiner [33] to confirm our manual findings.

Finally, an inter-rater agreement analysis was used to develop a classification scheme to categorize the identified refactorings under different technical debt categories. We also applied the Cohen's kappa coefficient [34] by reaching 0.86 as result between all the authors, which indicates high confidence of agreement. The few cases of disagreement were discussed between the authors at the end of the process and we were able to find a consensus for all of them.

IV. Results

A. Quantitative Analysis

We manually examined 193 unique commits from the different projects listed in Table III. The identification of refactorings in these commits and analyzing is a very labor-intensive manual task due to the lack of automated tools support. We found that 44 commits have Dockerfile-related refactoring, 51 commits have Docker-compose related refactorings, and 55 commits of regular code refactorings (e.g., Java, C++, etc.) due to changes in Dockerfile or Docker-compose. We observed

Authorized licensed use limited to: Université de Montréal. Downloaded on March 19,2023 at 18:04:54 UTC from IEEE Xplore. Restrictions apply.

TABLE IV: Discovered Docker-specific refactoring types

Refactoring Type	Artifact	Description
Extract stage	Dockerfile	Extract multi-stage building from a single-stage Building
Inline stage	Dockerfile	Aggregate multi-stage building into a single stage
Move stage	Dockerfile	Move a stage from a multi-stage context into another single stage context in a different Dockerfile
Sort Instructions	Dockerfile	Order Instructions sequence from the least frequently changing to the most frequently changing
Replace ADD Instruction with COPY Instruction	Dockerfile	Replace ADD instruction with COPY Instruction when files/directories need to be only copied from host to container
Extract Run Instructions	Dockerfile	Extract Run instructions commands into a separate script file
Inline Run Instructions	Dockerfile	Inline Run instructions into a single RUN instruction using && operator
Remove Run Instruction including mv command	Dockerfile	Delete RUN instruction with mv command and use previous COPY or ADD instructions to set the correct path
Update Base Image	Dockerfile	Avoid using unnecessary heavy image, replace base image with a lighter one
	Dockerfile	Avoid building and downloading dependencies on top of base image, replace base image with a larger one; if dependencies are fixed and a similar image exits in public or private repo,use an existing Image
Rename Image	Docker-compose	Set a relevant Name and TAG including the necessary information of your image (version, software..)
	Dockerfile	Add or rename Image alias
Update RUN Instruction	Dockerfile	Reformat RUN instruction commands; split commands into multiple lines where each line represents a single option/argument, order option/argument alphabetically, use backslash ...
Update Base Image TAG	Dockerfile	DRY principle, set a dynamic TAG using ARG instruction to avoid creating a new Dockerfile when TAG is only changing
	Dockerfile	Change TAG value or Replace latest TAG with an explicit TAG
Add ENV variable	Dockerfile	Add ENV variables to store useful system-wide values
	Docker-compose	
Add ARG instruction	Dockerfile	DRY principle, set dynamic instructions using ARG instruction to avoid creating a new Dockerfile for similar images
Extract Ports Attribute	Docker-compose	Extract ports attribute into an override Docker-compose file
Extract Volume Attribute	Docker-compose	Extract volume attribute into an override Docker-compose file
Extract ENV Attribute	Docker-compose	Extract ENV attribute into an override Docker-compose file or use .env file
Move Service	Docker-compose	Extract service into an override Docker-compose file
Rename Service	Docker-compose	Set a relevant Name for the service
Rename Container	Docker-compose	Set a relevant Name for the container
Rename Volume	Docker-compose	Set a relevant Name for the volume
Add Extends attribute	Docker-compose	Add Extends attribute to inherit configuration from an existing service thus avoiding duplication
Reorder Services	Docker-compose	Order services based on their dependency order
Update Image TAG	Docker-compose	Change TAG value or Replace latest TAG with an explicit TAG

a total of 43 commits containing false-positive (22%). The false-positive commits (i.e., keywords matching commits that did not include any refactorings), occurred due to different reasons, including (i) the keyword `refactor` was used in non-refactoring commit messages to highlight the needs for future refactorings; (ii) refactoring occurred out of Docker-specific context; or (iii) the lack of knowledge about the domain as Docker-specific refactorings are a new concept and were not explored before in the literature; thus some commit messages described regular changes that alternate the behavior as refactoring.

The identified Docker-specific technical debts are listed in Table V. For Dockerfiles, we found 6 technical debt categories related to *Image size*, *Build Time*, *Duplication*, *Maintainability*, *Understandability* and *Modularity*. For Docker-compose files, 3 out of these 6 Dockerfile categories are also applicable: *Duplication*, *Maintainability*, and *Understandability*. Furthermore, we found that *Extensibility* is another technical debt addressed by developers in Docker-compose files.

1) Docker-specific Technical Debts: Maintainability was a major technical debt target for applied refactorings in Docker projects representing 55 occurrences (38%) and found in Dockerfile and Docker-composed files. Indeed, container-based technologies are meant to provide the possibility to apply central modifications with having them rolled out over the system with small endeavors and no downtime. Furthermore, this technical debt was mainly associated mainly with the excessive usage of environmental variables (27 commits) and TAG updates (21 commits) in both Dockerfile and Docker-compose files.

Understandability is the major refactorings target in Docker-compose files. In this category, 75% of the refactorings were performed to improve naming. The observed renaming had variety of motivation, such as avoiding the usage of irrelevant names (e.g., using app as a service name), including software information (e.g., software version) in the image and TAG names, as well as keeping naming consistency throughout the project. Renaming was also present in Dockerfile (6 commits).

Build Time was among the least addressed debts in Dockerfile (0.14%). In fact, the order of the Dockerfile instructions highly matters when re-building, because when a step's cache is invalidated by changing files or modifying lines in the Dockerfile, subsequent steps of their cache will break. Thus, ordering steps from least to most frequently changing, is something to keep in mind when creating a Dockerfile in order to optimize caching.

Regarding to *Modularity*, we found 7 occurrences addressing only Dockerfiles where they try to apply multi-stage building by separating the build from the run-time environ-

Authorized licensed use limited to: Université de Montréal. Downloaded on March 19,2023 at 18:04:54 UTC from IEEE Xplore. Restrictions apply.

TABLE V: Discovered Docker-specific technical debt categories.

Technical Debt	Artifact	Goal	Situation	Consequence
Image size	Dockerfile	Refactorings are used to reduce the final image size	Adding new/changing components requires huge modifications and may lead to unexpected behaviors	Huge disk space, difficult to upload and a huge attack surface
Build Time	Dockerfile	Refactorings are used to reduce the build or rebuild time	Docker engine takes a lot of time to build the final image	Evolving and changing the image became difficult process
Extensibility	Docker-compose	Refactorings are performed to increase the level of abstractions and improve the reusability of the Dockerfile and Docker-compose files.	Adding new components require duplication	Duplication
Duplication	Docker-compose	Refactorings are introduced to fix duplicated fragments in Dockerfile and Docker-compose.	Duplicated fragments	Adding new/changing existing components is difficult and error-prone
Maintainability	Both	Refactorings are performed to ease the modification as well as preventing large impact/spread of future bugs/unexpected behavior	Adding new/changing components requires modifying existing files	Modifications and upgrades require huge endeavors and conceivable downtime
Understandability	Both	Refactorings are applied to reduce the effort to understand code, e.g., renaming elements	Huge efforts to understand code	Slight and simple modifications will become time consuming
Modularity	Dockerfile	Refactorings are used to reduce image complexity by breaking image into various stages.	Complex image	Adding new/changing existing components is difficult and error-prone

ment, which helps avoiding the inclusion of unnecessary build dependencies in the final image.

2) Docker-specific Refactoring Types: We organized the different refactoring types into a hierarchy based on the technical debt they addressed. Figure 1 shows the proposed taxonomy of the refactorings and their rationale. We have also highlighted the number of occurrence of these refactoring types in the analyzed commits. The gray square represents the refactoring type along with the number of occurrences, the blue ellipse represents the artifact (Dockerfile or Docker-compose), and the purple hexagon represents the technical debts which the detected refactoring addressed.

We categorized true-positive refactorings by manually classifying them into 24 new unique refactoring types. Table IV shows the lists of refactorings that were identified among the analyzed commits. We defined a total of 14 new Dockerfile-related refactoring types and 12 new Docker-compose ones as described in Table IV. The overall number of refactorings identified in the manual commits analysis is 146 to address the aforementioned technical debts.

We found three extra Dockerfile-related refactoring types performed to improve *Maintainability*, namely, *Update Base Image* (3 commits), *Extract RUN Instructions* (3 commits) and *Replace ADD Instruction with COPY Instruction* (4 commits). The first refactoring type consists of using existing images whenever it is possible, to avoid building and downloading dependencies on top of the base image. This will reduce maintainability efforts as all required installations are done and best practices are probably applied especially when using official images. The second refactoring type consists in extracting a shell script for a specific subsequent RUN instructions in Dockerfile. This refactoring types does not only shrinks the image size but also groups commands in a much more cleaner, simpler, and portable format. Finally, the third refactoring operation embraces in using COPY instruction instead of ADD instruction when files/directories need to be only copied from host to container.

It is important to notice that the ADD instruction supports other functionalities, such as the use of URL instead of a local files and it can also recognize the archive format and extract it directly into the destination. This additional functionalities might be considered as tricky in practice, as ADD instruction may behave extremely unpredictable. The result of such unreliable behavior often came down to copying when we want to extract and extracting when we want to copy. Unsurprisingly, a large proportion of the analyzed refactorings in Dockerfile aimed at reducing the image size (36%).

Regarding *Image Size*, we found five possible refactorings composed of *Extract RUN Instructions* (3 commits), *Inline RUN Instructions* (4 commits), *Remove RUN Instruction including MV command* (3 commits), *Update Base Image* (12 commits) and *Extract stage* (7 commits) were also included in this category. *Remove RUN Instruction including MV command*, in particular, aims at reducing layers number in attempt to shrinking the image size by removing the RUN instruction with MV command and using previous COPY or ADD instructions to set the correct path, if possible.

We also found several refactorings applied to make the Dockerfile less confusing and more modular. Such changes involved the improvement of structural aspects. For example, *Update RUN Instruction* (6 commits) in Dockerfile which consists in formatting commands within RUN instructions (e.g., splitting commands into multiple lines where each line represents a single option/argument) and *Reorder Services* (4 commits) in Docker-compose where developers reordered the services sequence based on their dependency order.

Unlike *Extract Stage*, *Inline Stage* (4 commits) and *Move Stage* (3 commits) aims at aggregating multi-stage building into a single stage when multi-staging is unnecessary (i.e., no significant dependencies) and extracting a stage from a multi-stage context into a new single-stage context (e.g., extracting test stage into a new Dockerfile), respectively. The reason behind these refactorings is to improve the build time when multi-staging can be avoided. As multi-staging requires building intermediary images that significantly affects the build time.

We also found in Dockerfile commits 10 refactorings aimed at improving code design by avoiding duplication and fostering

Authorized licensed use limited to: Université de Montréal. Downloaded on March 19,2023 at 18:04:54 UTC from IEEE Xplore. Restrictions apply.

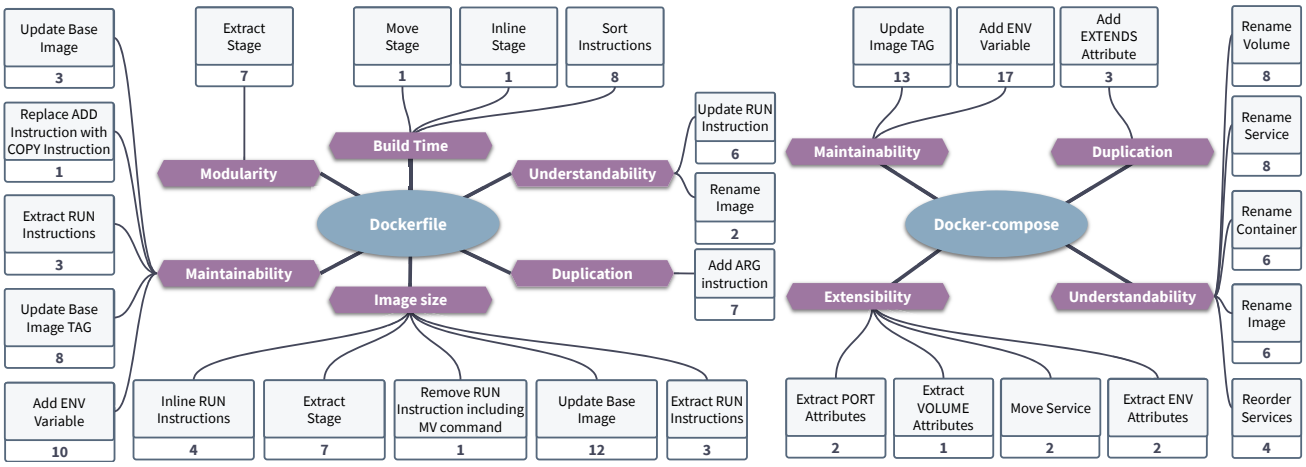


Fig. 1: Docker specific refactorings taxonomy.

the reuse of the code fragments. *Add ARG instruction* (7 commits) in Dockerfile involves adding arguments using the ARG instruction to define common identifiers (e.g., software version and file/directories paths), that can be later used inside other instructions such as COPY, ADD and RUN, leading to a dynamically changing Dockerfile, as these identifiers can be affected at build time. Besides, a total of 8 other Docker-compose related refactorings were found to make the configuration code more generalizable (3 commits), reusable (3 commits), and inter-operable (2 commits), by moving attributes to an override Docker-compose file. Such practice helps developers to reuse a single Docker-compose file across development and production while being able to run different services.

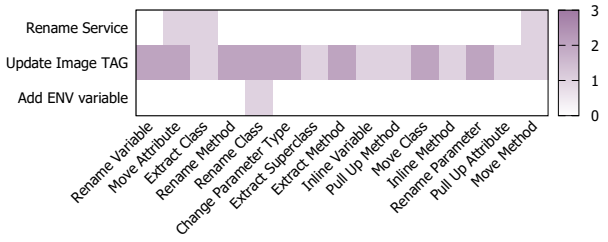
on the vertical axis and regular code refactorings are represented on the horizontal axis. Each cell in the heatmap indicates the number of occurrences of a Docker-specific refactoring type along with a regular code refactoring type. Darker colored cells indicates a strong co-occurence frequency while a lighter color signifies a weaker co-occurence. The observed refactorings included 22 regular code refactoring types along with 9 Docker-specific refactorings types, 3 for Docker-compose (a) and 6 for Dockerfile (b). It is clear that *Update Image TAG*, *Update Base Image* and *Extract Stage* are associated with extensive code refactoring of the hosting application involving almost all the refactoring types. In fact, those Docker-specific refactorings directly impact the code hosting the containers as they introduce significant changes in the Image or the Stage(s) which are typically called from the hosting code similar to functions in APIs in other contexts.

B. Qualitative Analysis

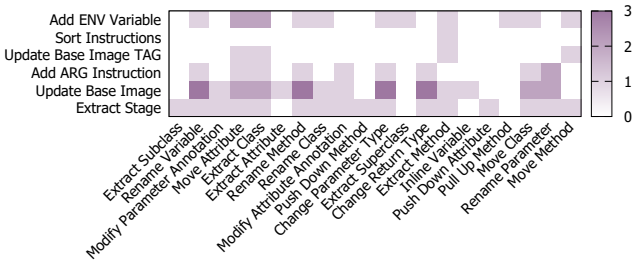
The qualitative analysis aims at obtaining and analyzing some examples of commits where the Docker-specific refactorings were found to address the most common/important technical debt issues discussed in the previous section.

1) Dockerfile Technical Debt and Refactoring Examples: Several of analyzed refactorings aimed at improving maintainability from several perspectives. Let us consider the example of the *Update Base Image* refactoring shown in Listing 4. In this listing, `openjdk` was initially used as a base image (Line 5) and `Dockerize` software was installed using RUN instructions (Lines 7–12). However, the official `Dockerize` image already exists with the required version v0.6.1 and it can be easily pulled from DockerHub to act as a starting point of the Dockerfile (Line 6). This can save a lot of time spent on maintenance because all the installation steps are done and official images can be highly trusted.

Build time is another critical technical debt in Docker projects. When working on evolving and changing the image, build time is considered a dead time. The *Sort Instructions*



(a) Docker-compose.



(b) Dockerfile.

Fig. 2: Refactorings co-evolution: regular code refactorings and Docker-specific refactorings.

Authorized licensed use limited to: Université de Montréal. Downloaded on March 19,2023 at 18:04:54 UTC from IEEE Xplore. Restrictions apply.

```
diff --git Dockerfile
--- a/docker/elasticsearch/Dockerfile
+++ b/docker/elasticsearch/Dockerfile
@@ -1,17 +1,12 @@
-FROM openjdk:8
+FROM jwilder/dockerize:0.6.1
-RUN apt-get update && apt-get install -y wget
-     && apt-get install -y curl
-ENV DOCKERIZE_VERSION v0.6.1
-RUN wget https://github.com/jwilder/docke[...]
-    && tar -C /usr/local/bin -xzvf docker[...]
-    && rm dockerize-linux-amd64-$DOCKERIZ[...]
```

Listing 4: Commit `4f221f9` from `datahub`: Refactored Dockerfile to reduce the number of pushed layers on a re-build.

refactoring presented in Listing 5 can address build time issues. In this example, a simple recompile of the app will change the `app.jar` file (Line 5) leading to a non-valid caching step when rebuilding the image, and the subsequent steps of the cache will then break (Lines 6–9). Ordering instructions from the least frequently changing to the most frequently changing will be highly efficient in such scenario. As mentioned in the commit message, the goal was to "*reduce the amount of pushed layers on a simple recompile*" leading to an optimize caching and a faster build time.

```
diff --git Dockerfile
--- [...]/src/main/docker/Dockerfile
+++ [...]/src/main/docker/Dockerfile
@@ -11,9 +11,10 @@
-ADD ifs-data-service-1.0-SNAPSHOT.jar app.jar
-RUN sh -c 'touch /app.jar'
-ENTRYPOINT ["java",[...],"-jar","/app.jar"]
-CMD curl -f http://localhost:8080/monito [...]
HEALTHCHECK --interval=10s --timeout=3s \
+CMD curl -f http://localhost:8080/monito [...]
+RUN sh -c 'touch /app.jar'
+ENTRYPOINT ["java",[...],"-jar","/app.jar"]
+ADD ifs-data-service-1.0-SNAPSHOT.jar app.jar
```

Listing 5: Commit `f7b5921` from `innovation-funding-service`: Refactored Dockerfile to reduce the number of pushed layer on re-build.

Image Size is the second most frequent category of technical debts in Dockerfile besides *Maintainbility*. Indeed, Docker projects are prone to image size increases due to layer based structure of the image. In general, the smaller the image, the quicker it is uploaded, and the faster it can scale. Besides, small images are considered to have less vulnerabilities. The *Update Image* refactoring shown in Listing 6 can used to fix issues related to the Image Size. In this example, `ubuntu:14.10` base image was replaced with `phusion/baseimage` and the subsequent RUN instructions (Lines 6–11) were extracted in a separate shell script called `install.sh`. This new base image is also an `ubuntu` based image, but it includes modifications for Docker-friendliness. Therefore, as each RUN instruction represents a unique and single layer, this refactoring can shrink the image by reducing the number of layers. Besides, the goal of this refactoring was also described in the commit message "*refactor of Dockerfile to generate images that occupy less space*".

```
diff --git Dockerfile
--- a/17.0-rc1/Dockerfile
+++ b/17.0-rc1/Dockerfile
-FROM ubuntu:14.10
+FROM phusion/baseimage
-RUN cd /usr/src \
-    && tar xf otp_src_${ERLANG_VERSION}.tar.gz\
-    && cd otp_src_${ERLANG_VERSION} \
-    && ./configure \
-    && make \
-    && make install
+RUN /build/install.sh
diff --git install.sh
--- /dev/null
+++ b/17.0-rc1/install.sh
```

Listing 6: Commit `9787e1a` from `docker-erlang`: Refactored Dockerfile to generate a smaller image.

2) Docker-compose Technical Debt and Refactoring Examples: Regarding to Docker-compose files, *Extensibility* is a common and frequent technical debt as described in the quantitative analysis. In general, the Docker-compose file's extensibility should be improved when both development and production environments are located in the same file. *Move Services* refactoring shown in Listing 7 can be used to improve the extensibility of Docker compose files. In this example developers moved `benchmark_resin` services (Lines 9–12) to a new override file (Line 16). Therefore, they were able to reuse a single Docker-compose file while being able to run different services.

```
diff --git docker-compose.yaml
--- a/docker-compose.yaml
+++ b/docker-compose.yaml
@@ -24,22 +24,3 @@ services:
-benchmark_resin:
-  build: {[...]}
-  depends_on: [...]
-  ports: [...]
diff --git/hexagon_benchmark/docker-compose.yaml
--- /dev/null
+++ b/hexagon_benchmark/docker-compose.yaml
```

Listing 7: Commit `99109b6` from `hexagon`: Refactored Docker-compose file to make "benchmark services" optional.

We have also found that the inheritance was mainly improved in Docker-compose files to remove *Duplication*. The *Add Extends Attribute* refactoring presented in Listing 8 is a common refactoring type for Docker-compose to address duplication issues. This refactoring helped to solve the configuration duplication issue by using the `extends` attribute (Line 12) and specifying the parent service `db`(from the parent Docker-compose file).

V. Implications and Discussions

A. Refactorings Co-Evolution

We found that several refactoring types when applied to Dockerfile and Docker-compose files impact the source code of the project that should be also refactored using regular refactorings as well. This co-evolution process is currently performed manually by developers and there is no semi-automate tool to support them. Thus, there are significant

Authorized licensed use limited to: Université de Montréal. Downloaded on March 19,2023 at 18:04:54 UTC from IEEE Xplore. Restrictions apply.

```
diff --git a/docker-compose-jasper.yml
--- a/docker-compose-jasper.yml
+++ b/docker-compose-jasper.yml
@@ -1,105 +1,61 @@
db:
-  image: muccg/postgres-ssl:9.4
-  environment:
-    - POSTGRES_USER=rdrfapp
-    - POSTGRES_PASSWORD=rdrfapp
-  ports:
-    - "5432"
+  extends:
+    file: docker-compose-common.yml
+    service: db
```

Listing 8: Commit `f7d2b4d` from `rdrf`: Refactored Docker-compose file to remove duplicated configuration.

costs that can be associated with Docker-specific refactorings which can make developers reluctant to apply them. This study identified the most common co-evolution patterns that we found in multiple commits and then a semi-automated tool can be designed to recommend code refactorings based on the applied Docker-specific refactorings.

B. Docker-specific Refactorings and Technical Debt

We observed in Section IV that, in practice, software developers applied several refactoring types at the Dockerfile and Docker-compose levels. Currently, all the identified refactoring types are manually applied due to the lack of any semi-automated tool support. With this proposed empirical study, we described the scientific foundations required to enable the implementation of the refactoring types identified for Dockerfiles and Docker-compose files. The same observation apply to the technical debt categories for Docker projects. Thus, tool builders can use the definition and symptoms discovered in this empirical foundation to define, validate, and implement detection rules to automatically identify the quality issues in Dockerfile and Docker-compose files. Such tools could not only save developers effort and time, but could also bring a discipline toward refactorings of Docker projects within a software development team.

Although we do not expect practitioners to fix all the detected technical debts independently from the context, we expect them to judge which quality issues in Dockerfiles and Docker-compose files are more relevant and adequate for their specific context. Team-leads can work with developers to establish customized guidelines from this study for dealing with technical debts in Docker projects. These guidelines could also trigger developers to capture the impact and the rationale of their Docker changes appropriately for each situation, by developing beneficial habits and long-lasting projects.

Our results can also provide a common ground for documenting, discussing, and assessing refactorings and their impact on Docker projects. This common ground will help educators to disseminate multiple dimensions of refactorings in Docker projects. Further, they could encourage the practice of refactoring continuously based on the examples and data provided in this empirical study.

C. Optimizing Docker Performance

It is not surprising that the main rationale of refactoring Docker projects is to optimize the usage of resources (e.g., memory, CPU, etc.) needed to execute the containers via reducing the image size, removing duplication and reducing the build time. Indeed, Docker projects are recently used extensively in cyber-physical systems including automotive industry and smart manufacturing [25], [20]. The hardware resources are typically limited, thus refactoring may play a bigger role in optimizing the size and performance of Docker projects once semi-automated tools are available based on the scientific foundation of this study.

VI. Threats to Validity

In this study, we selected 68 Docker projects to identify refactoring types and technical debt categories but they may not be representative to the very large number of Docker projects on GitHub. To address this threat, we ensured that the selected projects are diverse in domains, sizes and realistic in terms of the evolution history and popularity. Thus, we used various GitHub metrics including the number contributor, users, stars, commits, etc. to evaluate their popularity and diversity. Despite Java was the dominant language on applications hosting containers (selected to facilitate the manual analysis of the regular code refactoring), around 30% of the projects are written in C++. Indeed, the refactoring types (and their rationale) in object-oriented programming are almost the same, thus the impact of our selection criteria are limited to the generalizability of the co-evolution results between Docker and Docker-compose refactorings, and regular code ones.

The manual identifications and classification of refactoring types and technical debt categories can be subjective. To mitigate this threat, four experts evaluate all the selected commits separately and the agreement coefficient score between all of them was high for all the identification and classification results as discussed in Section III. For the few cases of disagreement, the different experts discussed the commit(s) after submitting their results to avoid any bias. Our study also involved many hours of manual inspection and analysis to understand and categorize the Docker-specific refactorings. To mitigate these threats, the four experts used the commit messages, pull-requests descriptions and comments in the code to better understand the context of the changes and they only marked the refactorings and their rationale when they are very confident about them. For the regular code refactorings (co-evolution study), RefMiner [33] was used to confirm and aid the manual inspection of the refactoring types, even though all commits were manually inspected carefully for any potential false positives.

VII. Related Work

A. Docker Smells

Few studies investigated quality issues in Docker projects and they are all limited to Dockerfile. Similar to traditional configuration code smells [29], Docker smells are indicators of certain designs flaws and weaknesses in the Dockerfile.

Authorized licensed use limited to: Université de Montréal. Downloaded on March 19,2023 at 18:04:54 UTC from IEEE Xplore. Restrictions apply.

Without being actual bugs, these smells potentially affect the image in a negative way. Although Docker's documentation provides a list of best practices[1] to fix different kind of smells, developers are still violating these recommendations. Yiwen et al. [36] divided Dockerfile smells into two major categories: DL-smells (referring to the violation of the official Dockerfile best practices rules) and SC-smells (referring to the violation of the basic shell scripts practices). They described Dockerfile smells occurrence by proposing an empirical study in open-source projects and they found that nearly 84% of these projects have smells in their Dockerfile code. Further, they found that DL-smells appear way more than SC-smells.

Some other studies focused on building automated and semi-automated tools to help the detection of bad practices in Dockerfile. Hadolint (Haskell Dockerfile Linter)[2], is a smart Dockerfile linter that can be used to help developers to build the best practice into the Docker images. The linter parses the Dockerfile into an AST and performs rules on top of the AST in order to detect DL-smells based on ShellCheck[3] to lint the Bash code inside RUN instructions for the SC-smells detection. Henkel et al. [15] proposed a tool (similar to Hadolint) named Binnacle, which performs rule mining over 178,000 Dockerfiles collected from GitHub. However, this tool mainly focused on bash related rules. Xu et al. [16] defined a new type of smell named TF-smell which stands for "Temporary file smell" it indicates a careless use of temporary file in image building process that may cause temporary file left in the image, which increases the image size and affects the distribution.

All above-mentioned studies focus on detecting bad practice/smells on Dockerfiles. None of them worked on suggesting possible refactorings or the technical debt categories, including smells, for Docker-compose files or the regular code refactoring observed in Docker projects.

B. Refactoring and Design Flaws

Our work is mainly related to 1) approaches identifying design flaws and recommending how to fix them; and 2) empirical studies on refactoring. Several approaches have been proposed to automatically detect design flaws (i.e., anti-patterns, code smells, etc.). We only discuss a few representative works and refer the interested reader to the recent survey by Sharma and Spinellis [30] for a complete overview. Marinescu [21] proposes a metric-based mechanism to capture deviations from good design principles and heuristics, called "detection strategies". Such strategies are based on the identification of *symptoms* characterizing a particular smell and *metrics* for measuring such symptoms. Moha et al. [23] exploit a similar idea in their DECOR approach, proposing a Domain-Specific Language (DSL) for specifying smells using high-level abstractions. Besides metrics exploiting structural information extracted from the code, Palomba et al. [26] provide evidence that historical data can be successfully exploited to identify code smells.

A lot of effort has been devoted to the definition of approaches supporting refactorings. One representative example is JDeodorant, a tool proposed by Tsantalis and Chatzigeorgiou et al. [32] able to detect and refactor the code to fix four code smells (i.e., *State Checking*, *Long Method*, *God Classes*, and *Code clones*). We point the interested reader to the survey by Bavota et al. [5] for an overview of approaches supporting code refactoring.

Empirical studies on software refactoring mainly aim to investigate software developers' refactoring habits and the relationship between refactorings and code quality. Murphy-Hill et al. [24] investigated how developers perform refactorings. Examples of the exploited datasets are usage data from 41 developers using the Eclipse environment and information extracted from versioning systems. Among their several findings, they show that developers often perform *floss refactoring*, namely they interleave refactorings with other programming activities, confirming that refactorings are rarely performed in isolation. Kim et al. [18] present a survey of software refactoring with 328 Microsoft's engineers to investigate when and how they refactor code and the developers' perception towards the benefits, risks, and challenges of refactoring [18]. They show that the major risk factor perceived by developers with regards to refactoring is the introduction of bugs and one of the main benefits they expect is to have fewer bugs in the future, thus indicating the usefulness of refactoring for code components exhibiting high fault-proneness.

To the best of our knowledge, this paper presents the first study on refactoring for Docker projects since most of the existing studies focus on traditional paradigms, such as object-oriented programming. Several of the aforementioned studies can be adopted to transfer the knowledge into this paradigm, but a first step is to provide an empirical foundation to define refactoring types for the Docker domain.

VIII. Conclusion

We proposed a study to advance the knowledge of Docker refactorings and technical debts, which are under-explored in the literature. We have manually investigated and defined specific refactoring types and technical debt categories for Docker projects. We have also studied the co-evolution between applying those Docker-specific refactorings and regular refactorings on the code of the app hosting the containers. A taxonomy of refactorings in Docker projects was proposed including 14 new Dockerfile-related refactorings, 12 Docker-compose related refactorings and 7 technical debt categories.

In the future, we will use the scientific foundations of this study to build new tools for refactorings detection and recommendation for Docker projects.

[1]https://docs.docker.com/develop/develop-images/dockerfile_best-practices
[2]https://github.com/hadolint/hadolint
[3]https://github.com/koalaman/shellcheck

References

[1] C. Abid, V. Alizadeh, M. Kessentini, T. do Nascimento Ferreira, and D. Dig. 30 years of software refactoring research:a systematic literature review. *IEEE Transactions on Software Engineering*, 1(1), 2020.

Authorized licensed use limited to: Université de Montréal. Downloaded on March 19,2023 at 18:04:54 UTC from IEEE Xplore. Restrictions apply.

[2] V. Alizadeh and M. Kessentini. Reducing interactive refactoring effort via clustering-based multi-objective search. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 464–474, 2018.

[3] V. Alizadeh, M. Kessentini, W. Mkaouer, M. Ocinneide, A. Ouni, and Y. Cai. An interactive and dynamic search-based approach to software refactoring recommendations. *IEEE Transactions on Software Engineering*, 2018.

[4] Anonymous Author(s). Study appendix, 2020. https://sites.google.com/view/ase21-docker-refactorings.

[5] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. Recommending refactoring operations in large software systems. In M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, editors, *Recommendation Systems in Software Engineering*, pages 387–419. Springer Berlin Heidelberg, 2014.

[6] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, page 47–52, New York, NY, USA, 2010. Association for Computing Machinery.

[7] 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 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR '17)*, pages 323–333, 2017.

[8] Z. Codabux and B. Williams. Managing technical debt: An industrial case study. In *4th International Workshop on Managing Technical Debt (MTD 2013)*, 2103.

[9] B. P. Datasets, 2020. https://cloud.google.com/bigquery/public-data.

[10] Docker, 2020. https://docker.com.

[11] Docker Compose, 2020. https://github.com/docker/compose.

[12] DockerHub, 2020. https://hub.docker.com.

[13] M. Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley Professional, 1 edition, 1999.

[14] 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*, pages 796–801, 2018.

[15] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps. Learning from, understanding, and supporting devops artifacts for docker. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE '20)*, 2020.

[16] iwei Xu, Y. Wu, Z. Lu, and T. Wang. Dockerfile tf smell detection based on dynamic and static analysis methods. In *Proceedings of the 43rd IEEE Annual Computer Software and Applications Conference (COMPSAC '19)*, 2019.

[17] 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*, 2015.

[18] M. Kim, T. Zimmermann, and N. Nagappan. An empirical study of refactoringchallenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, July 2014.

[19] P. S. Kochhar and D. Lo. Revisiting assert use in github projects. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering (EASE '17)*, pages 298–307, 2017.

[20] R. Lovas, A. Farkas, A. C. Marosi, S. Ács, J. Kovács, Á. Szalóki, and B. Kádár. Orchestrated platform for cyber-physical systems. *Complexity*, 2018, 2018.

[21] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the International Conference on Software Maintenance (ICSM '04)*, pages 350–359. IEEE, 2004.

[22] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni. Many-objective software remodularization using NSGA-III. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):17:1–17:45, 2015.

[23] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010.

[24] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2011.

[25] A. D. Neal, R. G. Sharpe, P. P. Conway, and A. A. West. smaRTI-a cyber-physical intelligent container for industry 4.0 manufacturing. *Journal of Manufacturing Systems*, 52:63–75, 2019.

[26] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489, 2015.

[27] Portworx. 2017 annual container adoption survey: Huge growth in containers, 2020. https://portworx.com/2017-container-adoption-survey/.

[28] N. Rachatasumrit and M. Kim. An empirical investigation into the impact of refactoring on regression testing. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 357–366. IEEE, 2012.

[29] T. Sharma, M. Fragkoulis, and D. Spinellis. Does your configuration code smell? In *Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR '16)*, pages 189–200. IEEE, 2016.

[30] T. Sharma and D. Spinellis. A survey on software smells. *Journal of Systems and Software*, 138:158 – 173, 2018.

[31] D. Silva, N. Tsantalis, and M. T. Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*, pages 858–870, 2016.

[32] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367, 2009.

[33] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 483–494, New York, NY, USA, 2018. ACM.

[34] A. J. Viera, J. M. Garrett, et al. Understanding interobserver agreement: the kappa statistic. *Fam med*, 37(5):360–363, 2005.

[35] Y. Wang, Y. Sun, Z. Lin, and J. Min. Container-based performance isolation for multi-tenant saas applications in micro-service architecture. *Journal of Physics: Conference Series*, 1486(5):052032, 2020.

[36] Y. Wu, Y. Zhang, T. Wang, and H. Wang. Characterizing the occurrence of dockerfile smells in open-source software: An empirical study. *IEEE Access*, 2020.

[37] L. Zhang, D. Tiwari, B. Morin, B. Baudry, and M. Monperrus. Automatic observability for dockerized java applications. *arXiv*, 248:1–14, 2019.

Authorized licensed use limited to: Université de Montréal. Downloaded on March 19,2023 at 18:04:54 UTC from IEEE Xplore. Restrictions apply.