An empirical study on self-admitted technical debt in Dockerfiles

Hideaki Azuma¹ · Shinsuke Matsumoto¹ · Yasutaka Kamei² · Shinji Kusumoto¹

Accepted: 1 November 2021 / Published online: 31 January 2022
© The Author(s) 2022

Abstract

In software development, ad hoc solutions that are intentionally implemented by developers are called self-admitted technical debt (SATD). Because the existence of SATD spreads poor implementations, it is necessary to remove it as soon as possible. Meanwhile, container virtualization has been attracting attention in recent years as a technology to support infrastructure such as servers. Currently, Docker is the de facto standard for container virtualization. In Docker, a file describing how to build a container (Dockerfile) is a set of procedural instructions; thus, it can be considered as a kind of source code. Moreover, because Docker is a relatively new technology, there are few developers who have accumulated good or bad practices for building Docker container. Hence, it is likely that Dockerfiles contain many SATDs, as is the case with general programming language source code analyzed in previous SATD studies. The goal of this paper is to categorize SATDs in Dockerfiles and to share knowledge with developers and researchers. To achieve this goal, we conducted a manual classification for SATDs in Dockerfile. We found that about 3.0% of the comments in Dockerfile are SATD. In addition, we have classified SATDs into five classes and eleven subclasses. Among them, there are some SATDs specific to Docker, such as SATDs for version fixing and for integrity check. The three most common classes of SATD were related to lowering maintainability, testing, and defects.

Communicated by: Vittorio Cortellessa

✉ Shinsuke Matsumoto
shinsuke@ist.osaka-u.ac.jp

Hideaki Azuma
h-azuma@ist.osaka-u.ac.jp

Yasutaka Kamei
kamei@ait.kyushu-u.ac.jp

Shinji Kusumoto
kusumoto@ist.osaka-u.ac.jp

¹ Graduate School of Information Science and Technology, Osaka University, Osaka, Japan

² Faculty of Information Science and Electrical Engineering, Kyushu University, Fukuoka, Japan

Keywords Self-admitted technical debt · Technical debt · Container virtualization · Docker · Infrastructure as code (IaC)

1 Introduction

Software developers often apply ad hoc solutions or workarounds to achieve their short-term implementation goals (Alves et al. 2014). Such solutions are called *technical debt* as a metaphor of *not-quite-right code which we postpone making it right* (Cunningham 1992). Among these ad hoc solutions, one intentionally introduced by a developer and explicitly noted in source code comments is called self-admitted technical debt (SATD). Well-known SATDs are often commented with TODO or FIXME. SATD is known to have a significant possibility of having negative impacts in the future (Fontana et al. 2012; Zazworka et al. 2011). Various SATD studies have been conducted on empirical study (Bavota and Russo 2016; Maldonado and Shihab 2015), automatic detection (Maldonado et al. 2017b), repayment (Maldonado et al. 2017a; Mensah et al. 2018), and domain-specific (i.e., DNN framework) SATDs (Liu et al. 2020).

Meanwhile, container-based virtualization has been attracting attention in recent years as a promising way to support infrastructure environments. Container virtualization has strong advantages in terms of less overhead and greater resource efficiency by sharing the host OS kernel compared with traditional hypervisor-based virtualization. Currently, Docker is the de facto standard for container virtualization. Docker is being used in over 87% of IT companies and many open-source software (Cito et al. 2017; Portworx 2020). One powerful benefit of using Docker is from the concept of *infrastructure as code (IaC)* (Humble and Farley 2010). The procedure of infrastructure setup can be explicitly described as a script file. This concept enables turning tacit infrastructure knowledge into explicit knowledge with documentation, automation of infrastructure management, and version management.

Our empirical study is motivated by the assumption that *various SATDs must exist in Dockerfile as they do in general programming languages*. The Dockerfile, a source code file for a Docker image,¹ is described as a set of procedural instructions. Because software developers often write code that functionally works but includes quality problems, it can be naturally considered that developers of Docker images may introduce *not-quite-right code* into their Dockerfiles with comments. Additionally, Docker is a relatively new technology released in 2013. Therefore, if we collect and analyze such SATDs from many Dockerfiles, we can acquire practical knowledge of typical concerns of and workarounds by Docker developers. This knowledge can be referred to as *bad practices in Docker*. Once we know what these bad practices are, we can detect low-quality Dockerfiles that contain bad practices. Previous study results for comprehending SATD in general open-source software (Potdar and Shihab 2014) have been applied to other studies of automatic SATD detection (Maldonado et al. 2017b) and repayment (Maldonado et al. 2017a). Thus, it is expected that the results of our SATD analysis will also be able to be applied to automatically detect Docker SATDs and to obtain patterns of SATD repayment.

In addition, SATDs in Dockerfile may spread to other extended Docker images because Docker supports a mechanism for image inheritance. For example, when we deploy a custom web server with Node.js, the server can be easily created by extending the Node.js image using a FROM node instruction in our Dockerfile. Therefore, SATDs in the Node.js

¹ A Docker image is an immutable snapshot of a container.

image will affect our newly created web server. To avoid such diffusion of low-quality implementations, it is necessary to investigate and detect SATD.

In this paper, we describe an empirical study to better understand SATD in Dockerfiles. Our main research questions are as follows:

RQ1: How many SATDs are present in Docker?

RQ2: What types of SATD exist in Dockerfiles?

RQ3: What is the distribution and frequency pattern of SATD in Dockerfiles?

First, we examined comments from published Dockerfiles using Docker Hub and GitHub. Then, the first three authors conducted a manual inspection to classify them using pattern-based SATD detection proposed in previous SATD work (Potdar and Shihab 2014). After the consensus formation process, we defined a new classification for Docker SATD, including some Docker-specific SATD types.

Our manual classification was conducted for 382 comments in Dockerfiles, which were collected from the top 1,250 images of Docker Hub.² The results showed that many SATDs exist in Dockerfiles, and 3.4% of all comments are about SATD. Also, we classified Docker SATD into five classes and eleven subclasses. This classification includes Docker-specific SATD, such as integrity checking and image size reduction. Since we conduct a mixed approach from a large pool of comments and manually inspect 382 samples, we believe that our work significantly contributes to the validation of empirical knowledge about SATD (i.e., 3.4% SATD based on the results that are derived from 382 samples that are originally from 2,364 comments) in the field of containerization projects. The remainder of this paper is organized as follows. Section 2 provides preliminaries for our study. Section 3 describes the methodology of our empirical study. Section 4 presents the results for each research question. Section 5 discusses our results and implications for researchers and developers. Section 6 discusses related work. Section 7 discloses the threats to the validity of our study. Finally, Section 8 presents our concluding remarks.

2 Preliminaries

2.1 Technical Debt

In 1993, Cunningham introduced a metaphor, technical debt, to describe an ad hoc solution for programming problems (Cunningham 1992). Since this introduction, many researchers have studied technical debt (Brown et al. 2010; Kruchten et al. 2012; Tom et al. 2013).

Potdar and Shihab (2014) investigated the technical debt intentionally implemented by a developer and explicitly noted in source code comments. In this research, they used the label SATD and used source code comments as an indicator of SATD to analyze how many SATDs existed in projects. In their results, SATD existed in 2.4% to 31% of the files from four open-source projects. Bavota and Russo (2016) conducted a replication study of Potdar et al.'s work. They found 273 SATDs and classified them into six classes and ten subclasses. The classes are Code debt, Design debt, Documentation debt, Defect debt, Test debt, and Requirement debt. Each class, except for the Test debt, has two subclasses. Liu et al. (2020) investigated SATD in DNN frameworks and found 7,159 SATDs in 7 DNN frameworks. They also identified two domain-specific SATD classes in their study: Compatibility debt

²<https://hub.docker.com/>

and Algorithm debt. Compatibility debt refers to debt related to a project's dependencies on other, immature projects, which cannot supply all qualified services. Algorithm debt corresponds to sub-optimal implementations of algorithm logic in the DNN framework. These domain-specific SATDs ranked in the top three most common SATD categories in some DNN frameworks. Many studies have shown that SATD harms software (Wehaibi et al. 2016; Zazworka et al. 2011). Therefore, a wide variety of studies are still underway to comprehend (Maldonado and Shihab 2015; Zampetti et al. 2018), detect (Huang et al. 2017; Maldonado et al. 2017b), and repay (Maldonado et al. 2017a; Mensah et al. 2018) of SATDs.

2.2 Container Virtualization

Container virtualization provides a virtual environment called a container. Since Docker, the current de facto standard for container virtualization (Open Container Initiative 2020), was released, container virtualization has rapidly been attracting attention (Cito et al. 2017).

One of Docker's remarkable benefits is its ability to describe the container construction process as a script. Further, Docker has a feature to build a Docker image from a Dockerfile, a text file in script format. Hence, the procedure of infrastructure setup can be explicitly described. The other benefit of Docker is reducing resource overhead by sharing the host OS kernel (Docker 2020), which allows efficient management of resources.

Moreover, Docker allows us to create a new image based on an existing image. Using this feature, Docker users can create customized Docker images of the base image to suit the environment in which their application will run.

2.3 SATD in Docker

To motivate our work, we show an example of a Dockerfile containing a typical SATD. This SATD was introduced in February 2019 on Docker Compose project³ and is slightly simplified for readability.

```

1 FROM centos:latest
2 RUN apt-get update && apt-get install -y \
3     gcc \
4     python2 \
5     ...
6
7 # FIXME: virtualenv 16.3.0 breaks build, force 16.2.0 until fixed
8 RUN pip install virtualenv==16.2.0
9 RUN pip install tox==2.9.1
10 ...
11
12 COPY entrypoint.sh /usr/local/bin/
13 ENTRYPOINT ["/usr/local/bin/entrypoint.sh"]

```

Usually, Dockerfile starts with an image name with FROM instruction to specify its base or parent image. This example uses the latest version of CentOS as a base image. Using the RUN instruction, we can freely execute any type of command on the image. The example invokes the package manager command, apt-get, to install gcc, python2, and other packages. Now, we can see a SATD-related comment depending on an external library, named virtualenv. The developer of this image pointed out that a specific version of virtualenv was

³<https://github.com/docker/compose/commit/f472fd5>

breaking for some reason, so the developer decided to use an older version as a workaround and leave a comment with `FIXME` attention. Our study collects these ad hoc solutions (i.e., SATDs) from Dockerfile comments.

Java projects have usually been used as research subjects in previous SATD studies (Bavota and Russo 2016; Wehaibi et al. 2016). In these studies, researchers have pointed out that SATDs diffuse to other classes and projects due to object-oriented inheritance. Because Docker has a feature to inherit arbitrary images, the problem of SATD diffusion may occur in Dockerfiles as well. Furthermore, although there have been studies on SATD for specific domains, such as DNN frameworks (Liu et al. 2020), no studies have focused on Dockerfiles. Therefore, it is an important issue to clarify the nature of SATD in Dockerfile and prevent diffusion.

The motivation for each of our three research questions is listed below.

RQ1: How many SATDs are present in Docker?

Motivation: Previous studies indicated that SATDs exist in source code written in Java, C++, and other languages. However, to the best of our knowledge, none of the existing studies examined SATD in Dockerfiles. Therefore, we should know how many SATDs exist in Dockerfiles.

RQ2: What types of SATD exist in Dockerfiles?

Motivation: If SATD exists in Dockerfiles, a solution is needed, but one solution will not work for all SATDs. That is, different types of SATD require different solutions. Therefore, we investigated what types of SATD exist in Dockerfiles.

RQ3: What is the distribution and frequency pattern of SATD in Dockerfiles?

Motivation: Docker image developers are likely to have trouble with the problem referred to in each SATD comment. It is considered that many developers are annoyed by the SATDs that are most frequently identified in RQ2. Accordingly, we quantified each type of SATD to gain knowledge on which type of SATD we need to pay more attention to.

3 Methodology

This section describes our investigation methodology, including dataset construction and our manual classification for SATD. Figure 1 shows the flow of the investigation methodology. The top part of Fig. 1 shows how the dataset was constructed starting from Docker Hub (Section 3.1). The bottom part shows the flow of manual classification (Section 3.2). Each box in the figure shows the dataset composition at each step of the process. The box color varies according to the type of data, and the numbers in parentheses are the numbers of repositories or comments collected. In the following, we explain the investigation methodology according to the flow of this figure.

3.1 Dataset

3.1.1 Selection of Target Projects

In this study, because we examined comments in Dockerfiles, we needed a dataset that consisted of Dockerfile comments, which are often added during the development process (Fluri et al. 2007). It is assumed that SATDs are also added during the development process.

Fig. 1 Overview of investigation methodology

Fig. 2 Distribution of SATD-like comments in Docker images

Hence, we analyzed the Dockerfiles for projects that are being continuously developed and maintained. Specifically, we constructed the dataset from Dockerfiles in popular repositories of Docker Hub. The reason for this is that the main part of Docker Hub repositories is Docker images and Dockerfiles that are likely to be well developed. Moreover, popular images in Docker Hub are often inherited. If SATDs are left in a popular image, it is assumed that many images will be adversely affected by them. Therefore, the priority of SATD repayment in popular images of Docker Hub is considered to be high.

However, Docker Hub repositories do not store Dockerfiles directly. In many cases, external services such as GitHub are used to control those versions. Docker Hub repositories often have URLs pointing to their external repositories. In this study, we collected links to GitHub repositories associated with the 1,250 most popular Docker Hub repositories. The 1,250 images were retrieved from 50 Docker Hub pages, which include 25 images per page. The reason for limiting the number of pages to 50 is due to the distribution of SATD-like comments in Docker images. The distribution is shown in Fig. 2. The x-axis shows the Docker Hub page sorted by popularity, and the y-axis shows the number of SATDs.⁴ The distribution follows Pareto's principle. The first top 10 pages (i.e., 250 images) account for almost 80% of the SATD-like comments. Thus, we concluded that 50 pages are sufficient to conduct our empirical study. As a result, we collected 462 GitHub repositories from 1,250 images. We only found 462 repositories from 1,250 projects because some repositories refer to the same repository, and the others repositories do not refer to their source code repository. Then, we collected all Dockerfiles (3,149 in total) contained in all 462 repositories. Finally, we extracted comments from the Dockerfiles to obtain 12,694 comments. To extract these comments, we used a JavaScript library for syntax highlighting called highlight.js.⁵ When it is applied to the Dockerfiles, an html tag `<span class="hljs-comment"></span>` is added to each comment section. Therefore, by extracting only the parts with this tag, we could extract comments from the Dockerfiles. Comments that were written consecutively over several lines were treated as a single comment.

3.1.2 Merging Comments

Usually, software projects have almost the same Dockerfiles to support different base images (e.g., alpine, buster, or slim) or different software versions (Java 8 or Java 11).

⁴SATDs are detected by simple keyword matching. Details are shown in Phase 1, Section 3.2

⁵<https://highlightjs.org/>

This fact might be a strong duplication bias in the study. On the other hand, a single Docker Hub project does not always refer to a single Dockerfile. Docker can be used for multiple purposes in software development. For example, the Apache Spark project uses several Dockerfiles for release management⁶ and several testings.^{7,8,9} These file contents are totally different and may contain different debts. Therefore, we decided to collect all Dockerfiles from all GitHub repositories referred to by Docker Hub projects to cover various technical debts. To eliminate the duplication bias, we remove duplicate comments which have the same comment body. As a result of this aggregation, the total number of comments in the dataset was reduced to 2,364.

3.1.3 Removal of Unnecessary Comments

Dockerfile contains comments that are clearly not SATDs, such as license comments and comments inserted by automatic generation tools (autogenerated comments). Hence, we removed the license and autogenerated comments from the dataset using simple keyword matching with `license`, `copyright` and `autogenerated`. The first author then manually confirmed that those comments were, in fact, license or autogenerated comments. As a result, 121 comments were removed, yielding 2,364 comments for classification.¹⁰

3.2 Manual Classification

In this study, the first three authors conducted a manual classification to investigate how many SATDs exist in Dockerfiles and what types of SATD are present in Dockerfile. In this section, we describe the manual classification process performed by the three authors.

3.2.1 Phase 1: Consensus Building for SATD Classification

Before classifying SATDs in Dockerfile, it was necessary to understand what comments were classified as SATD in the previous studies and to build a consensus among classifiers on the classification criteria for each SATD. To acquire the knowledge and deepen our understanding of SATDs, we used a dataset created by Maldonado and Shihab (2015) that categorizes Java comments. The dataset was formed by comments collected from ten Java projects classified into five types of SATD and Non-debt classes. The five types of SATD are Design debt, Defect debt, Test debt, Requirement debt, and Documentation debt. In Phase 1, we randomly extracted a total of 100 comments from the dataset, 20 for each type of SATD and Non-debt. Each author picked comments that they did not understand the reason for the comments' classification. Then, the three authors discuss these comments and created clear criteria for each SATD based on the discussion.

⁶<https://github.com/apache/spark/tree/master/dev/create-release/spark-rm/Dockerfile>

⁷<https://github.com/apache/spark/tree/master/external/docker/spark-test/base/Dockerfile>

⁸<https://github.com/apache/spark/tree/master/external/docker/spark-test/worker/Dockerfile>

⁹<https://github.com/apache/spark/tree/master/external/docker/spark-test/master/Dockerfile>

¹⁰While the GitHub repositories that are targeted in our study have eight Dockerfiles on average (3,149 Dockerfiles/462 repositories), general open-source software projects have many more source code files (e.g., Table I in the work by Maldonado and Shihab (2015) shows at least 1,000 Java classes per project). Therefore, the total number of comments is smaller than such previous studies that work on SATD.

3.2.2 Phase 2: Applying SATD Classification to Docker Domain and Defining Classification Criteria

Before conducting manual classification, 2,364 comments were automatically segregated according to whether they were likely to contain SATD. The purpose of this was twofold. One was to manually classify the comments that are most likely to be SATD on a priority basis. The other purpose was to extend the consensus-building of SATD on Java in Phase 1 to Docker. This automatic classification was based on whether the comments contained a SATD pattern. The SATD pattern was found by Potdar and Shihab (2014) containing 63 phrases, which is usually contained in SATD-related comments, such as `FIXME`, `hack`, `ugly`, and `may cause problem`.¹¹ This pattern-based comment classification is simply conducted using the `grep` command. In the results, 52 comments included the SATD pattern, and 2,312 comments were determined not to include it. In Phase 2, a total of 100 comments were classified by the three authors independently. Of the 100 comments, 50 had the SATD pattern and 50 did not. The authors discussed the results of the classification in each phase. Through the discussion, the three authors unanimously decided on a single class for each comment. After discussion, it became clear that the SATD classification criteria of Maldonado and Shihab (2015) do not fully fit the comments in the Dockerfile dataset. Therefore, we defined classification criteria that better fit the Dockerfile comments based on the classification of Bavota and Russo (2016).

3.2.3 Phase 3: Manual Classification of SATD in Dockerfiles

In Phase 3, further manual classification was applied for more comments to improve the generalizability of the study. The classification target includes 2 (52 – 50) remaining comments having SATD patterns. Also, we expand the classified target for comments without the SATD pattern to improve the investigation coverage. However, since the total number of remaining SATD pattern-free comments was 2,262 (2,312 – 50), manual classification of all these comments requires a great deal of time and effort. Similarly to Bavota and Russo (2016), we decided to classify 330 randomly selected comments from this set, which represents a 95% confidence level sample with a 5% confidence interval. In this phase, we first tried to classify 100 of the 330 comments. Consequently, the total number of classification targets in this phase was 102 (2 + 100) comments. Based on the classification results in Phase 3, the classification criteria were adjusted through discussion.

3.2.4 Phase 4: Reclassification and Completion of Classification

Finally, of the 330 comments without the SATD pattern, 230 comments that had not been classified yet were classified independently by the three authors. In Phase 4, as in Phase 3, the classification criteria were adjusted through discussions. The final classification criteria arrived at in this phase are shown in Table 1. Note that some SATDs were assigned to a class that is not based on the final classification criteria. Therefore, the first author visually confirmed all SATDs, and then the three authors discussed and reclassified those comments.

¹¹For more details, the string `TODO` was added to the 63 patterns. We added `TODO` as an important keyword for discriminating SATD, although it is not included in the 63 patterns.

Table 1 Definitions and examples of subclasses

Subclass		
Definition		
Example		ID
Code/Workaround		
SATDs on not the best implementation		
# FIXME: Renaming manifest.xml is a workaround for (...) misbehaving when both manifest.xml and package.xml are present.		1357
Code/MissingFunctionality		
SATDs on the lack of functionality inside containers		1777
# TODO: Run both pip and pip3 via virtualenvs		
Code/BaseImage		
SATDs on the base image		
# TODO: Switch to Debian buster once the bug is resolved in a clean way.		297
Code/Version		
SATDs on fixing to a specific version		1613
# TODO: Pin versions for (url).		
Test/IntegrityCheck		
SATDs on integrity check		623
# TODO: Add PGP checking when the feature will be added to build system		
Test/ImprovementForTest		
SATDs on improvement for testing		
# FIXME: Some of this is only really needed for testing, it would be nice to split this up		754
Defect/Workaround		
SATDs on working around for existing bugs in external systems		737
# FIXME: Virtualenv 16.3.0 breaks build, force to use 16.2.0 until fixed.		
Defect/LatentBug		
SATDs on latent bugs		
# In PHP 7.4+, the pecl/pear installers are officially deprecated and will be removed in PHP 8+		538
Design/SizeReduction		
SATDs on size reduction of Docker image		
# TODO: This is here because extras binaries are copied from \$PATH into the bundles dir. It would be nice to handle better.		760

Table 1 (continued)

Subclass	
Definition	
Example	ID
Process/Deployment	
SATDs on deployment of the Docker image	
# By default you'll get a single-node server that stores everything in RAM. Don't use this configuration for production.	877
Process/Review	
SATDs on review of the Dockerfile	
# TODO: Figure out what they means by their bare "arm" arch.	1652
Unclassifiable	
Obviously SATD, but unclassifiable	281
# TODO: aufs-tools	

4 Study Results

The purpose of this section is to answer the three research questions from the point of view of objective facts. Implications and insights are discussed in the next section.

4.1 RQ1: How Many SATDs are Present in Docker?

Figure 3 shows the proportion of SATD in manually classified comments.¹² The left pie chart shows the proportion of SATD in comments with the SATD pattern. The right chart shows the proportion of SATD in comments without the SATD pattern. The number in the middle of each chart represents the number of manually classified comments. Unidentifiable comments, which exist only in the right chart, refer to comments that could not be determined as SATD or not due to the ambiguity of the content. Some comments could not be judged as SATD only using the content of the comment and the code before and after them. We judged them in addition on information such as the history of version control and website which its URL was inserted in the code. Of the comments that were manually classified, 45 (86.5%) comments with the SATD pattern were SATDs. In contrast, five (1.5%) comments without the SATD pattern were SATDs. If the 2,312 comments without the SATD pattern included SATDs in a similar proportion, it is considered that about 35 of these comments are SATDs. Thus, we can estimate that there are about 80 SATDs in the total of 2,364 comments for classification, which is about 3.4%.

The results of RQ1 yielded the following a finding.

Finding 1: The proportion of SATD in Dockerfiles is about 3.4%.

¹²The comment population does not include non-debt-related comments (i.e., license and auto-generated sentences) because they surely do not contain SATDs.

Fig. 3 Proportion of SATD in manually classified comments

4.2 RQ2: What Types of SATD Exist in Dockerfiles?

In the manual classification, 50 SATDs were classified into five classes and eleven subclasses. These classification criteria were created by mutual agreement of the three authors who conducted the manual classification based on the classification of Bavota and Russo (2016). Figure 4 shows the classification tree and the results of our manual classification. The six top nodes below the primary “SATD” node in the Fig. 4 represent the classes. Below these are the eleven leaves that represent subclasses of the classes. In addition, the number in the circle at the upper right corner of each node represents the number of SATDs in each class or subclass. Table 1 shows the definitions and examples of the subclasses applied in

Fig. 4 Our defined classification tree for Docker SATDs. The number in the circle at the upper right of each node indicates the number of instances

this study. Note that the examples in the table are not original texts of the comments found in the manual classification but have been modified to express the definitions clearly. The ID column shows the ID of each example in our dataset. Because our data are available on Google Sheets, readers can obtain detailed information about each comment by referring to its ID.¹³ The details of these subclasses are described below.

Code debt: Code debt reduces the maintainability of source code. In this study, Code debt is classified into four subclasses: Workaround (10 instances), Missing functionality (7 instances), Base image (4 instances), and Version (5 instances).

Code/Workaround refers to compromised implementation and is the most common subclass. This subclass includes many comments stating that the code should be improved later because the developer is aware of the existence of more optimal ways but has retained a compromised implementation due to time constraints or other factors. Other SATDs in this subclass tend to be expressions that it is not clear whether there is a better method, but if there is a better method, developers would like to change to that method. The subclass “Workaround” also exists in Defect debt described below, but SATDs in the Code/Workaround subclass are not a reference to a workaround for bugs but rather a temporary implementation.

Code/MissingFunctionality refers to a lack of functionality inside containers that cannot be observed from their outside. These SATDs refer to functionalities that are ancillary and have a low priority for implementation, rather than features that will be fatal if not implemented. For example, there is a comment about wanting to make the Python package manager work with Virtualenv.

Code/BaseImage alerts other developers to bugs in the base image or asks for changes to the base image. Some SATDs include both the former and latter elements. Such an SATD states that developers want to use a certain image as a base, but it has a bug. Hence, developers must use another image until the bug is fixed. In the comment, it is also stated that they want to change the base image when the bug is fixed. Because this subclass is related to the Docker-specific concept of the base image, it is considered to be a Docker-specific SATD.

Code/Version refers to version fixation of the frameworks or tools retrieved by package managers and Git. The SATD in this subclass is likely to exist because Docker officially recommends fixing the software version to be downloaded in the container as an official best practice. In this subclass, there are comments that not only encourage developers to fix a particular version but also encourages them always to get the latest version.

Test debt: Test debt comments ask for tests and container verifications. In this paper, Test debts are classified into two subclasses: Integrity check (8 instances) and Improvement for test (1 instances).

Test/IntegrityCheck refers to the lack of an integrity check on binary files or hash values used in a container. Usually, such an integrity check is conducted by Pretty Good Privacy (PGP), GNU Privacy Guard (GnuPG), or Linux sha256sum commands. Because Docker often requires external files, it is assumed that many of these SATDs exist in Dockerfiles. Therefore, like the Code/Base image subclass, this subclass is also a Docker-specific SATD.

Test/ImprovementForTest comments ask for improvements in testing methods. The improvements are defined not as fixing bugs in tests but as improving test efficiency and

¹³The original text and more information about each example are included in our dataset available at <https://docs.google.com/spreadsheets/d/1ZCkdLxQjYzpp88NtXYcSCNko8HX-2-uUzX217pf67s>.

maintainability. In the manual classification, three SATDs were assigned to this subclass. Developers attempt to build multiple images with a single Dockerfile in all of the Dockerfiles having these SATDs. The comments suggest that it will be better to manage the building of images for testing in another Dockerfile.

Defect debt: Defect debt describes bugs whose complete resolution has been postponed due to time limitations or low priority. In this paper, Defect debts are classified into two subclasses: Workaround (4 instances) and Latent (2 instances).

Defect/Workaround refers to measures to avoid bugs in external systems, which are often used in quantity by Docker containers. If the systems contain bugs, developers need to take measures to avoid the bug. In some cases, developers temporarily modify the environment of an image that has an SATD to deal with the bug. Therefore, images that have these SATDs require careful treatment. In contrast to Code/Workaround, SATDs in this subclass are about workarounds related to bugs or failures.

Defect/LatentBug indicates that an image has latent or future bugs. The bugs mentioned in these SATDs do not adversely affect the image as it was built. However, updates of external systems without backward compatibility will make the currently used commands unusable, and bugs will occur. We identify this SATD subclass by mentions of certain features being unavailable in PHP versions 8 and above.

Design debt: Design debt refers to implementations against design patterns. Because the domain-specific language (DSL) used in Dockerfile is not an object-oriented language like Java, its design pattern is different. Therefore, there are no subclasses that exactly matched subclasses of the classification made by Bavota and Russo (2016) in Java. All the Design debts found in this study are classified as Size reduction (2 instances).

Design/SizeReduction seeks a better implementation method to reduce the image size. Docker recommends keeping the image size as small as possible to reduce the cost of pulling and pushing images (Documentation 2020). Thus, we consider that these SATDs exist when we find comments suggesting that unwanted binaries and redundant copies exist in the container. Because SATDs in this subclass do not directly affect execution, it is considered that size reduction is often postponed.

Process debt: Process debt refers to problems in a specific process of development, such as deployment or Dockerfile review. These SATDs are closely related to the infrastructure tool aspect of Docker, such as deployment. Therefore, it is a Docker-specific SATD that does not exist in the classification by Bavota et al. In this paper, Process debts are classified into two subclasses: Deployment (2 instances) and Review (2 instances).

Process/Deployment refers to problems that occur in deployment. As described in Section 2.2, Docker is used as an infrastructure tool in many software developments. In addition, there are many developers who deploy their applications in Docker containers, and some use the same container for both the development and production environments. Thus, it is considered that special care will be required for deployment. The Process/Deployment SATD comments cautioned against using the default configuration in the production environment.

Process/Review asks for a review of the Dockerfile itself. These SATDs do not request a review for verification of external systems, but a review of the Dockerfile or the image itself by other developers. Because Docker is a relatively new technology released in 2013, developers may not have accumulated Docker know-how yet, leading to this SATD.

Unclassifiable debt: Unclassifiable debt is considered to be SATD because these comments request something, but we cannot assign these SATDs to any of the other classes

due to the ambiguity of the description. All of these SATDs contain the string `TODO`, which indicates a demand for something. In our classification, although the authors tried to understand the meaning of the comments by tracing the history of version control, they were unable to understand the purpose of the required future action.

The results suggest that the SATDs in Dockerfile can be classified into five classes and eleven subclasses, excluding Unclassifiable debt. In addition, we confirmed the existence of Docker-specific SATDs that do not exist in other languages and domains. Therefore, the following findings are obtained as results of RQ2.

Finding 2: There are five classes and eleven subclasses of Docker SATD.

Finding 3: Docker-specific SATDs exist.

4.3 RQ3: What is the Percentage of Each SATD?

Figure 5 shows the proportion of each SATD class and subclass. The percentages were obtained by dividing the number of SATDs belonging to each class and subclass by 50, the total number of SATDs found in this study. The upper band represents the classes, and the lower band represents the subclasses. The colors for each SATD class and subclass are the same as in Fig. 4.

The results show that Code debt is the most common class, accounting for 52.0% of all SATDs, followed by Test debt at 18.0%. The results of SATD classification for Java by Maldonado and Shihab (2015) showed that Test debt accounts for about 2% of the total SATD, whereas in Dockerfiles, it accounts for nearly 20%. Thus, the implementation of testing is likely more often put off in Docker.

Fig. 5 Proportion of each SATD class and subclass

The most common subclass is Code/Workaround, which accounts for 20.0% of all SATDs, followed by Code/MissingFunctionality and Test/IntegrityCheck. Therefore, it is considered that many image developers are concerned about the optimal implementation methods of various features and integrity checks in Docker. In addition, the total proportion of Docker-specific SATDs is 42.0%. The Docker-specific subclasses are Code/BaseImage, Code/Version, Test/IntegrityCheck, Design/SizeReduction, and Process/Deployment.

As a result of RQ3, the following findings are obtained.

Finding 4: Code debt and Test debt are common SATDs in Dockerfiles.

Finding 5: Docker-specific SATDs account for about 42% of all SATDs in Docker.

5 Discussion

In this section, we discuss the results of our manual classification by comparing it to the results of a manual classification in Java by Maldonado and Shihab (2015). Also we show implications for researchers, developers of Docker, and developers of Docker images.

5.1 Additional Analysis for Project Types

This section conducts a further analysis focusing on the type of Docker project. Docker Hub projects can be classified into official images¹⁴ and community images. Official images are published and maintained by Docker or a commercial entity. This means that these images are considered to be well maintained by highly experienced developers. The others images are called community images. There are several findings by SATD studies: highly experienced developers tend to introduce more SATDs (Potdar and Shihab 2014), and SATD increases over time in a system (Bavota and Russo 2016). Therefore, our assumption is that official images contain more SATDs compared with community images.

Within our collected 1,250 projects, 449 (35.9%) were official images. In total, 1,397 comments of 2,364 subject comments (59.0%) were extracted from officials. Additionally, 31 of 50 SATD comments (62.0%) were from officials. In other words, official images contains more SATDs compared with community images. We conclude that highly experienced and longer-lived Docker projects contain more SATDs, as indicated by existing SATD studies.

5.2 Comparison with SATD Classification in General Programming Language

This section discusses our defined SATD classification (RQ2) while comparing the original classification provided by Bavota and Russo (2016). The original classification is shown in Fig. 6. At a bird's eye view, the four parent classes (i.e., code, test, defect, and design) have the same definition as the original definition. One class (i.e., process) is found only in our

¹⁴For more details, image types defined by Docker Hub are official and verified images. An official image is published by Docker itself, and a verified image is maintained by a commercial entity. This paper calls them official images.

Fig. 6 Original SATD classification tree defined by Bavota and Russo (2016)

definition, and the another class (i.e., requirement) exists only in the original definition. In the following, we discuss a more detailed difference.

Code debt has two Docker-specific subclasses: Code/BaseImage and Code/Version. These two SATDs have surely occurred from fundamental features of Docker.

As mentioned in Section 4.3, SATD in Docker is considered to have a higher Test debt proportion than SATD in Java (RQ3). One of the reasons for this is that there are many testing frameworks for Java, but there are few testing frameworks for Docker. Docker requires a variety of tests and verifications, such as integrity checks of externally obtained binary files and testing whether an external system works with the image. Moreover, there is more than one method to do an integrity check, and these methods require various software, such as PGP and GnuPG, depending on the target files. Therefore, we consider that there is a difficulty in testing Docker containers that do not exist in the testing source code written in general programming languages such as Java.

Although the original classification contains no subclass in the test SATD, our classification defines two subclasses: integrity check and improvement. Similar to the base image and version, integrity checking is one of the important processes in Dockerfile from a security perspective. A concrete example of integrity checking SATD is "Add PGP checking when the feature will be added to build system". We consider that integrity checking is often postponed because it is a non-functional requirement and it is not necessary to provide functional features in a Docker image. Furthermore, Test/Integrity-Check SATDs are only found in Dockerfiles, which are included in the top 250 most popular Docker images in Docker Hub. Since most of these images are official Docker repositories, it is considered that few non-official image developers recognize the importance of integrity checks. The integrity check prevents man-in-the-middle attacks during the download of binary files. Hence, it is necessary to inform many developers about the importance of the integrity check efficiently. For example, if there is a system that automatically complements the integrity check part of the command by just providing a URL, many developers will readily recognize its importance. We consider that such a system can be realized by using a linter for Dockerfiles called Hadolint (Martinelli 2021). In addition, if there were an integrated development environment for Dockerfiles that implemented such a system, the quality of the Docker images may be further improved.

Next, we focus on Defect debt. Defect debt in general programming languages often refers to a bug within the code. In contrast, because Docker requires many external systems and tools for building a Docker image, Defect debts in Dockerfiles often refer to a bug in external systems and tools. It is considered that SATDs such as Defect/LatentBug exist in Dockerfiles because it is susceptible to external factors that are beyond the control of the image developers.

The original classification defines code smells and design patterns as a subclass of design debt. For example, Bavota and Russo (2016) showed that the "FIXME extract some

method" is one of the code smell debts. However, we could not confirm any of these debts. We consider this is because there is no well-known and named practical knowledge in Dockerfile. Instead, we found Docker-specific design debt, and size reduction. A smaller image size is good practice in Dockerfile and often indicates an improvement.

We merged requirement debt into code debt because the differences in these debts are not clear in Dockerfile. For example, Bavota and Russo (2016) indicated that one of the requirement debt is "The system doesn't have a function that allows retrieval of a sequence of attribute values". We agree that this example is a requirement debt because it describes *what to do* rather than *how to do it*. In Dockerfile, there are several SATDs, which we could not determine to be requirement debts. For instance, the comment, "TODO: Install Apache/Nginx for plugin development", explicitly says *what to do* but it also implicitly says *how to do it*. In order to satisfy the requirement, a package-manager command need only be called with Nginx. Therefore, we decided to include SATDs relating missing functionality into one of the code debts.

Meanwhile, we were not able to find any Documentation debt, probably due to the lack of documentation frameworks in Docker. While Java supports many documentation frameworks, such as Javadoc and Doxygen, Docker supports few documentation frameworks. In addition, through our manual classification, we found that developers tend to give an explanation of what each instruction does in the comment for that instruction. Therefore, it is unlikely that Documentation debt will occur in Docker. The absence of a documentation framework in Docker may be related to the RQ1 findings in this paper, where the SATD in Dockerfiles is about 3.4%. In Dockerfiles, developers often include a single-line comment for almost each shell instruction to show their intent explicitly. This is because the shell syntax is definitely imperative, not declarative. Therefore, it is considered that the rate of comments for the total number of lines of code (comment rate) is higher than for other languages, such as Java. To investigate this, we compared the comment rate per project in the ten Java projects used in the study by Maldonado and Shihab (2015) with the comment rate per GitHub repository used in this paper. The medians of each domain are 5.6% for Java and 12.1% for Docker, clearly showing that Docker has a higher comment rate. This may lead to a relatively low rate of SATD.

5.3 Accuracy of Pattern-Based SATD Detection

Next, we evaluated the usefulness of the SATD pattern for SATD identification. Two types of automatically classified SATDs were found in this study: true positives (tp) and false negatives (fn). True positives are SATDs that have been correctly classified by the automatic classification with the SATD pattern. Although false negatives are also SATDs, they were not correctly classified by the automatic classification. The Non-debt comments were classified as false positives (fp) or true negatives (tn). False positives are Non-debt comments that were classified as SATD by the automatic classification. True negatives are Non-debt comments that were correctly classified by the automatic classification. The number of true positives, false negatives, false positives, and true negatives is as follows. First, based on the results of the manual classification for the 52 comments with the SATD pattern, there are 45 true positives and 7 false positives. Based on the assumption mentioned in Section 4.1, there are approximately 35 SATDs in total in the comments without the SATD pattern. Therefore, there are 36 false negatives. The usefulness of the SATD pattern for identification can be evaluated by precision ($P = \frac{tp}{tp+fp}$), recall ($R = \frac{tp}{tp+fn}$), and F1 measure

($F = 2 \times \frac{P \times R}{P + R}$). The precision and recall are 86.5% (45/52) and 56.3% (45/(45 + 35)), respectively, yielding an F1 measure of 0.68. Thus, we can use the SATD pattern to identify SATD in Dockerfiles with high accuracy.

5.4 Implications

First, we discuss the implications and suggestions for Docker and SATD researchers. Our analysis shows that there are several types of Docker-specific SATDs (RQ3) and that there are many SATDs in Dockerfiles (RQ1). According to the SATD survey paper reported by Sierra et al. (2019), SATD studies are broadly classified into three categories; detection, comprehension, and repayment. We should conduct further empirical research into Docker-specific SATDs regarding these three categories. Specifically, the impact of SATDs in Dockerfile should be studied as a SATD comprehension issue because of the inheritance mechanism of Docker images. Detecting SATDs in a parent image and reporting the impact thereof might be practical information for Docker image developers.

In addition, we are making our dataset available to the research community. Our dataset consists of 2,364 comments, including 382 comments classified for Dockerfiles. Each comment in the dataset has a link to its source code, Dockerfile path, revision, and content. Thus, our dataset can be used to conduct further SATD analysis or replication studies. Moreover, SATD-related information, including revision history and Dockerfile paths, will make it easier to trace the version control history.

Next, we discuss implications for Docker developers. Our study reveals a problem that may be troubling to many Docker image developers. As described in Section 4.3, many developers tend to have problems with optimal implementation and integrity checking. Therefore, we consider that official documents, such as best practices, should indicate model cases of the best implementation methodologies and the need for integrity checks in a way that will be seen by many developers. As shown in Section 5.2, Docker lacks a function to convey the specification of the image as a document. Thus, we consider that a documentation tool for Docker will make it easier to understand the image specifications without reading the comments of each instruction.

Finally, we discuss implications for Docker image developers. The results showed that common types of SATD in Dockerfiles are Code debt and Test debt (RQ3). This gives image developers a clearer view of what they need to prioritize when building Docker images. Because developers do not know what types of SATD exist in Docker, they may have mixed SATDs. However, it is expected that our findings will help developers to avoid SATD contamination consciously. This will prevent not only the addition of new SATD but also prevent the diffusion of existing SATD to other images.

6 Related Work

6.1 Software Engineering in SATD

In the field of software engineering, many studies have been conducted about design pattern violations and code smells. These *bad patterns* can be described as one kind of technical debt. For example, Zazworka et al. (2014) analyzed four indicators to identify technical debt. They used modularity violations, absence of design patterns, code smells, and bug issues as indicators of technical debt. Comparing the four indicators, they found that different technical debts can be found in each of them and that the overlap between the different

debts detected by the four indicators is small. Fontana et al. (2015) analyzed the prioritization of code debt using code smell. Specifically, they used JCodeOdor, a tool for detecting code smells, to identify the most critical code smells to prioritize problems. Technical debt includes any implementation that will adversely affect future software development and maintenance, regardless of the developer's perception. Many SATD studies have explicitly suggested the existence of debt placed in the code deliberately by developers.

First, Potdar and Shihab (2014) defined SATD as the technical debt that is recognized by developers. They investigated how many SATDs existed in five open-source projects and found that these projects contained SATDs in the range of 2.4% to 31.0% of the total number of comments per file. Furthermore, in their study, they found 62 patterns that are likely to be described with SATD. Our manual classification was based on their 62 SATD patterns. Although the existence of SATDs was analyzed, it was not clear what type of SATD existed. Maldonado and Shihab (2015) classified SATD in ten Java projects. They found that, out of 33,093 comments, 2,676 described SATDs and classified them into five classes. Their results show that Design debt and Requirement debt are the most common types of SATDs in projects. Although our study also found some Design debt in Docker, these debts only indicated the need for size reduction of the Docker image. The class hierarchy is an important aspect of object-oriented languages, whereas Docker does not support such a design mechanism. This might be a reason why the proportion of Design debt found in our study is small, as Requirement debt suggests the presence of unfinished code parts. Our study could not identify this debt because Docker containers are required to be as small as possible and have one feature. In a general programming language, a source code file often has plural features. Bavota and Russo (2016) conducted a replication study of the study by Maldonado et al. and classified the SATD of large projects. In their study, they merged the ontology of technical debt developed by Alves et al. (2014) and the classification of Maldonado et al. to create new classification criteria. They also classified SATDs into subclasses, which were more detailed than the five classes established by Maldonado et al. Our classification criteria are based on their criteria.

In SATD studies, manual classification by the authors is the primary method of classifying SATD. Although some studies have classified SATD using automatic detection, such as natural language processing (Maldonado et al. 2017b), they can only be used in domains where the SATD that exists is already known. In our study, we classified the SATD in Dockerfiles by manual classification because it had not been analyzed yet.

In many previous studies, researchers focused on object-oriented languages, such as Java, because of a large number of static analysis tools for code and a large number of open-source projects (Bavota and Russo 2016; Maldonado and Shihab 2015; Huang et al. 2017). For example, in the study of SATD classification by Maldonado and Shihab (2015), they investigated ten Java open-source projects. Meanwhile, Liu et al. (2020) investigated SATD in a specific domain, the DNN framework, rather than in object-oriented languages. The results of their study showed that two types of SATD specific to the DNN framework have been discovered, suggesting that focusing on one domain may lead to the discovery of specific types of SATD. In Docker, images are built with a script-formatted file, a Dockerfile, which is written using DSL. Because this DSL is different in nature from other common programming languages, it was considered that new types of SATD are likely to exist in Dockerfiles.

In addition, Docker has its own specific practices. For example, it is recommended that images have a property called idempotency (Documentation 2020). This property ensures

that the result does not change no matter how many times it is run to improve the reproducibility of runtime events. Furthermore, developers are recommended to minimize the layers in Docker images (Documentation 2020), which reduces the cost of image pulling and pushing to the registry. To achieve this practice, multiple RUN instructions are combined into a single RUN instruction as much as possible. Thus, because Docker has its specific practices, it is expected that Docker SATD is also greatly influenced by these practices. Our study focused on a new domain called Docker and investigated its SATD.

6.2 Software Engineering in Docker

As described in Section 2.2, Docker is now the de facto standard for container virtualization. Docker is used by a wide variety of companies, with 87% of IT companies reporting that they are running Docker containers (Portworx 2020). It is also a trendy research area, with more than 7,770 research papers published on Docker since the beginning of 2020.¹⁵

In software engineering, a wide variety of Docker studies have investigated specific aspects, including the ecosystem of Docker containers (Cito et al. 2017), Docker build failure (Yiwen et al. 2020), and other Docker topics (Henkel et al. 2020a, b). These studies aimed to examine practical findings and lessons learned by mining a large number of published Dockerfiles. For example, Zhang et al. (2018) mined Dockerfiles from 2,840 projects and investigated the evolutionary trajectories of the Dockerfiles. They provided researchers and developers with the following findings: we should use official Docker images and reduce the number of image layers to improve image quality. Cito et al. (2017) conducted an empirical study on 70,000 Dockerfiles to characterize the Docker ecosystem. They contrasted their dataset with samplings containing the top 100 and top 1,000 most popular Docker-using projects. They showed that popular images are changed more often than the other images, with an average of 5.81 revisions per year. Our study also mined Dockerfiles, and we aimed to make the details of our empirical study available to many developers and researchers.

Similar to our study, some researchers have conducted empirical analysis from the aspect of risk for Docker images published on Docker Hub. Shu et al. (2017) proposed a Docker image vulnerability analysis framework to analyze the security vulnerabilities of Docker images. They conducted a large-scale study of security vulnerabilities in both official and community images on Docker Hub using the framework. Their results showed that child images inherit on average 80 or more vulnerabilities from their parents. Our study is partially motivated by these analysis results. SATD should be analyzed and identified to prevent SATD propagation the same as other vulnerabilities. Zerouali et al. (2019) conducted a study on security vulnerabilities and bugs in outdated Docker containers. They found that nearly half of the vulnerabilities in Docker containers had not been fixed, and all containers they studied used packages that contain bugs. While they focused on security vulnerabilities and bugs of Docker containers, we focused on various types of SATDs beyond security vulnerabilities.

While many studies focusing on Docker have been conducted (Haque et al. 2020; Oumaziz et al. 2019), as far as we know, there has been no study on SATD in the Docker domain. SATD may be one of the causes by which image quality deteriorates if developers do not repay the debt. Moreover, the developers' distress over SATD can be seen in the

¹⁵<https://scholar.google.com/scholar> (accessed 2020-10-18)

comments in Dockerfiles. Therefore, in our study, we analyzed the SATD in Dockerfiles and aimed to provide findings to contribute to its repayment and prevention.

6.3 Software Engineering in Infrastructure as Code

Docker is one of the tools that can achieve IaC, in addition to other tools such as Puppet (Krum et al. 2014), Chef (Taylor and Vargo 2014), and Ansible (Red Hat 2021). Various studies on IaC have been conducted (Hummer et al. 2013; Jiang and Adams 2015). For example, Sharma et al. (2016) studied code smells in the configuration files of Puppet, a tool that automates OS configuration and application building. They proposed a catalog of 24 configuration smells, including an *incomplete tasks* smell. Existence of this smell means that the code contains FIXME or TODO tags indicating incomplete tasks. Their results indicate that the *incomplete tasks* smell was found in more than 1/4 of the total projects they examined. This means that, even in IaC scripts, many developers indicate incomplete parts in their code with strings such as TODO. Based on this result, we considered that SATDs are also described in Dockerfiles with TODO comments.

Rahman et al. (2019) proposed a static analysis tool called “Security Linter for Infrastructure as Code,” which detects security smells in Puppet scripts. Their code can detect seven types of smells, including *suspicious comment* and *use of weak cryptography algorithms* smells. The *suspicious comment* smell means that a script has comments that describe potential problems and defects. The *use of weak cryptography algorithms* smell means that a script uses low-security cryptography algorithms, such as MD5 and SHA-1. These smells are similar to SATD in Dockerfiles, which indicates that SATD in Dockerfiles is related to technical debt in other IaC scripts.

Various researchers are working on technical debt expressed as code smell in IaC scripts. However, to the best of our knowledge, SATD in Dockerfiles has not been studied yet, and our study is the first attempt to understand debt in that domain.

7 Threats to Validity

7.1 Construct Validity

For the sake of simplicity, we decided to make our classification exclusive, which is similar to existing SATD studies (Potdar and Shihab 2014; Bavota and Russo 2016). This decision will affect the classification results. For example, the following comment (#2044¹⁶) can be considered to belong to two classes: Code/Workaround and Code/MissingFunctionality.

```
1 # TODO: write a tool to make writing Go module-friendly
   Dockerfiles easier.
2 RUN go install cloud.google.com/go/storage
3 RUN go install golang.org/x/build/autocertcache
4 RUN go install ...
```

If a SATD comment seems to belong to two or more classes, then we set priorities for these classes and take the highest-priority class. For comment #2044, the developer indicated that a useful Go-friendly feature is required for *an external tool*. Then, a naive

¹⁶<https://github.com/cloudfoundry/bosh-deployment-resource/blob/c57369/vendor/golang.org/x/net/http2/h2demo/Dockerfile#L16-L17>

approach (i.e., constantly call `go-install` command) was taken as *a workaround in Dockerfile itself*. In this case, we set a higher priority for a workaround because the missing feature is required for the external system, not the Dockerfile itself.

7.2 Internal Validity

Because we classified SATD manually, our understanding of Docker and its SATD increased as the classification progressed. Therefore, it is possible that the classification criteria gradually changed. To reduce the potential for this effect, the classification was conducted in phases. After each classification phase was completed, we discussed the results and adjusted the classification criteria when we agreed that changes were needed. Considering that adjustments to the classification criteria might change the classes or subclasses of SATD classified in the previous phases, the first author visually checked them after all classifications were completed. For the SATDs that were considered to need reclassification, the three authors discussed and reclassified them in phases. However, because the classification was conducted manually, our subjectivity may have influenced the classification.

Also, the internal validity of the study would be threatened by the dataset extraction, especially in comment merging and pattern-based SATD collection. Extracted comments are merged based on textual matching. If two debt comments have a few textual differences but almost the same contents, then they are treated as different debts. Furthermore, SATDs must exist in SATD pattern-free comments. Actually, our estimation, described in Section 4, showed that 1.5% (5/330) of SATD pattern-free comments are SATDs. This means we possibly missed 36 SATDs in the study by the statistical estimation. These SATDs might affect the classification results.

In our study, we found 50 SATDs, obtained through the manual classification of 382 comments. As a result, five classes and eleven subclasses were obtained. Because our classification system resulted from only 50 SATDs, we may discover new classes and subclasses or SATDs belonging to classes not found in our study but found in previous studies by classifying more comments.

7.3 External Validity

Due to the nature of our study, which aimed to reveal the reality of SATD in the Docker domain, it was necessary to collect Dockerfiles with ample comments. Such Dockerfiles are likely to be well-developed. In addition, due to the inheritance of Docker images, SATD in Dockerfiles has a risk of spreading to many images (Shu et al. 2017). Because the more popular a Docker image is, the more likely it is to be inherited, it is considered that repaying the SATDs of popular images is urgent. Therefore, we only investigated the Dockerfiles that are used to build the most popular images in Docker Hub. As such, our study results may be biased to some extent because the data were collected from only Docker Hub. It is possible that our results are common to official Docker images but not to general Docker images. Selection of a different Dockerfile dataset may change our results slightly. For example, it is possible to collect Dockerfiles from well-maintained projects with a large number of comments. Another way is to use a tool such as Hadolint (Martinelli 2021) to measure the quality of Dockerfiles of common projects and include those that meet certain criteria in the dataset.

Data can also be collected directly from popular repositories on GitHub. However, it is considered that popular repositories on GitHub are those where tools and applications are evaluated. Because a Dockerfile only has the scope of an infrastructure tool, it is likely to

be out of the scope of evaluation in most cases. Therefore, we did not collect data directly from the GitHub repository.

Moreover, it is likely to be pointed out that, because we collected data only from the most popular images in Docker Hub, our results included only a small absolute number of SATDs. However, including Dockerfiles that build a lower-ranked image in Docker Hub could result in a large amount of data that do not meet the aforementioned criteria of being well maintained and developed. Furthermore, after counting the number of SATDs found in our study for each image, it is clear that most of them are among the top 250 most popular images in Docker Hub. Therefore, the absolute number of SATDs may not increase greatly by augmenting the dataset with Dockerfiles that build lower-ranked images.

8 Conclusion

In this study, we analyzed SATD in the comments of Dockerfiles to investigate the extent of SATD in the Docker platform. As a result, we found that SATD exists in Dockerfiles, as it does in other general programming languages. About 3.4% of the total Dockerfile comments were identified as SATDs. These SATDs were classified into five classes and eleven subclasses, including Docker-specific subclasses.

In the future, we have a plan to conduct a qualitative study in which we ask Docker developers that wrote the studied Dockerfiles about their opinions on the different types of SATDs found in Dockerfiles. Also, we plan to develop automatic SATD detection for Dockerfiles that does not rely on the SATD pattern. This study serves as the first step in understanding SATDs in Docker, but the repayment of these debts has not been clarified yet. We also believe that studying SATD repayment is an important issue because it can be expected to help developers of Docker images by providing suggested repayment patterns.

Replication Our collected dataset is available online¹⁷ to facilitate future work.

Acknowledgements This research was partially supported by JSPS KAKENHI JP18H03222 and JP21H04877.

Declarations

Conflict of Interest The authors have no relevant financial or non-financial interests to disclose. The authors have no competing interests to declare that are relevant to the content of this article. All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript. The authors have no financial or proprietary interests in any material discussed in this article.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

¹⁷<https://docs.google.com/spreadsheets/d/1ZCkdLxQjJyZpp88NtXYcSCNko8HX-2-uUzX217pf67s>

References

- Alves NSR, Ribeiro LF, Caires V, Mendes TS, Spínola RO (2014) Towards an ontology of terms on technical debt. In: Proceedings of the international workshop on managing technical debt, pp 1–7
- Bavota G, Russo B (2016) A large-scale empirical study on self-admitted technical debt. In: Proceedings of IEEE/ACM working conference on mining software repositories, pp 315–326
- Brown N, Cai Y et al (2010) Managing technical debt in software-reliant systems. In: Proceedings of FSE/SDP workshop on future of software engineering research, pp 47–52
- Cito J, Schermann G, Wittern JE, Leitner P, Zumberi S, Gall HC (2017) An empirical analysis of the docker container ecosystem on github. In: Proceedings of international conference on mining software repositories, pp 323–333
- Cunningham W (1992) The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger* 4(2):29–30
- Docker (2020) Docker overview. <https://docs.docker.com/get-started/overview/>. Accessed 28 Oct 2020
- Documentation D (2020) Best practices for writing dockerfiles. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/. Accessed 17 Sept 2020
- Fluri B, Wursch M, Gall HC (2007) Do code and comments co-evolve? On the relation between source code and comment changes. In: Proceedings of working conference on reverse engineering, pp 70–79
- Fontana FA, Ferme V, Spinelli S (2012) Investigating the impact of code smells debt on quality code evaluation. In: Proceedings of the international workshop on managing technical debt, pp 15–22
- Fontana FA, Ferme V, Zanoni M, Roveda R (2015) Towards a prioritization of code debt: a code smell intensity index. In: Proceedings of the international workshop on managing technical debt, pp 16–24
- Haque MU, Iwaya LH, Babar MA (2020) Challenges in docker development: a large-scale study using stack overflow. In: Proceedings of the international symposium on empirical software engineering and measurement (ESEM), pp 1–11
- Henkel J, Bird C, Lahiri SK, Reys T (2020a) A dataset of dockerfiles. In: Proceedings of the international conference on mining software repositories, pp 528–532
- Henkel J, Bird C, Lahiri SK, Reys T (2020b) Learning from, understanding, and supporting devops artifacts for docker. In: Proceedings of the international conference on software engineering, pp 38–49
- Huang Q, Shihab E, Xia X, Lo D, Li S (2017) Identifying self-admitted technical debt in open source projects using text mining. *Empir Softw Eng* 23(1):418–451
- Humble D, Farley D (2010) Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education
- Hummer W, Rosenberg F, Oliveira F, Eilam T (2013) Testing idempotence for infrastructure as code. In: Proceedings of the international conference on distributed systems platforms and open distributed processing, pp 368–388
- Jiang Y, Adams B (2015) Co-evolution of infrastructure and source code—an empirical study. In: Proceedings of the working conference on mining software repositories, pp 45–55
- Kruchten P, Nord RL, Ozkaya I (2012) Technical debt: from metaphor to theory and practice. *IEEE Softw* 29(6):18–21
- Krum S, Hevelingen WV, Kero B, Turnbull J, McCune J (2014) Pro Puppet. Apress
- Liu J, Huang Q, Xia X, Shihab E, Lo D, Li S (2020) Is using deep learning frameworks free? Characterizing technical debt in deep learning frameworks. In: Proceedings of the international conference on software engineering, pp 1–10
- Maldonado EDS, Shihab E (2015) Detecting and quantifying different types of self-admitted technical debt. In: Proceedings of the international workshop on managing technical debt, pp 9–15
- Maldonado EDS, Abdalkareem R, Shihab E, Serebrenik A (2017a) An empirical study on the removal of self-admitted technical debt. In: Proceedings of the IEEE international conference on software maintenance and evolution, pp 238–248
- Maldonado EDS, Shihab E, Tsantalis N (2017b) Using natural language processing to automatically detect self-admitted technical debt. *IEEE Trans Softw Eng* 43(11):1044–1062
- Martinelli L (2021) Haskell dockerfile linter. <https://github.com/hadolint/hadolint>. Accessed 18 Jan 2021
- Mensah S, Keung J, Svajlenko J, Bennin KE, Mi Q (2018) On the value of a prioritization scheme for resolving self-admitted technical debt. *J Syst Softw* 135(C):37–54
- Open Container Initiative (2020) Why have all of these companies come together? <https://opencontainers.org/faq/#why-have-all-of-these-companies-come-together>. Accessed 8 Sep 2020
- Oumaziz MA, Falleri J, Blanc X, Bissyandè TF, Klein J (2019) Handling duplicates in dockerfiles families: learning from experts. In: Proceedings of the IEEE international conference on software maintenance and evolution, pp 524–535

- Portworx (2020) Annual container adoption report. <https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf>. Accessed 5 Oct 2020
- Potdar A, Shihab E (2014) An exploratory study on self-admitted technical debt. In: Proceedings of the IEEE international conference on software maintenance and evolution, pp 91–100
- Rahman A, Parnin C, Williams L (2019) The seven sins: security smells in infrastructure as code scripts. In: Proceedings of the international conference on software engineering, pp 164–175
- Red Hat (2021) Ansible. <https://www.ansible.com/>. Accessed 25 Jan 2021
- Sharma T, Fragkoulis M, Spinellis D (2016) Does your configuration code smell? In: Proceedings of working conference on mining software repositories, pp 189–200
- Shu R, Gu X, Enck W (2017) A study of security vulnerabilities on docker hub. In: Proceedings of the conference on data and application security and privacy, pp 269–280
- Sierra G, Shihab E, Kamei Y (2019) A survey of self-admitted technical debt. *J Syst Softw* 152:70–82
- Taylor M, Vargo S (2014) Learning chef: a guide to configuration management and automation. O'Reilly Media Inc
- Tom E, Aurum A, Vidgen R (2013) An exploration of technical debt. *J Syst Softw* 86(6):1498–1516
- Wehaibi S, Shihab E, Guerrouj L (2016) Examining the impact of self-admitted technical debt on software quality. In: Proceedings of the international conference on software analysis, evolution, and reengineering, pp 179–188
- Yiwen W, Yang Z, Tao W, Huaimin W (2020) An empirical study of build failures in the docker context. In: Proceedings of the international conference on mining software repositories, pp 76–80
- Zampetti F, Serebrenik A, Penta MD (2018) Was self-admitted technical debt removal a real removal? an in-depth perspective. In: Proceedings of the international conference on mining software repositories, pp 526–536
- Zazworka N, Shaw MA, Shull F, Seaman C (2011) Investigating the impact of design debt on software quality. In: Proceedings of workshop on managing technical debt, pp 17–23
- Zazworka N, Vetro' A, Izurieta C, Wong S, Cai Y, Seaman C, Shull F (2014) Comparing four approaches for technical debt identification. *Softw Qual J* 22(3):403–426
- Zerouali A, Mens T, Robles G, Gonzalez-Barahona JM (2019) On the relation between outdated docker containers, severity vulnerabilities, and bugs. In: Proceedings of the international conference on software analysis, evolution and reengineering, pp 491–501
- Zhang Y, Yin G, Wang T, Yu Y, Wang H (2018) An insight into the impact of dockerfile evolutionary trajectories on quality and latency. In: Proceedings of annual computer software and applications conference, pp 138–143

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.