Master Thesis

Title

An Empirical Study on Self-Admitted Technical Debt in Container Virtualization

Supervisor
Prof. Shinji KUSUMOTO

by
Hideaki Azuma

February 2, 2021

Departmant of Computer Science
Graduate School of Information Science and Technology
Osaka University

Master Thesis

An Empirical Study on Self-Admitted Technical Debt in Container Virtualization

Hideaki Azuma

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 Docker know-how. 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 thesis is to systematize SATDs in Dockerfiles and to share knowledge with developers and researchers. To achieve this goal, we conducted a manual classification for SATDs in Dockerfile. Since SATDs are described in source code comments, we need a dataset that consists of comments from Dockerfiles. We have constructed the dataset from Dockerfiles in the top 1,250 most popular repositories of Docker Hub. Although the dataset contains 2,907 comments, it is difficult to analyze all comments manually. Therefore, we used SATD patterns found in previous studies to select the targets of manual classification. As a result, out of the total 2,907 comments in our dataset, 405 comments are subjected to the classification. The classification is independently conducted by the author and two faculty members.

We found that about 3.4% of the comments in Dockerfile are SATD. In addition, we classified the SATDs into five classes and eleven subclasses. Among the classes, there are some Docker-specific SATD classes, such as SATDs for version fixing and for integrity check using Pretty Good Privacy commands.

Keywords

Self-admitted technical debt
Technical debt
Container virtualization
Docker
Docker Hub
Infrastructure as code

Contents

1 Introduction 1

2 Preliminaries 3

2.1 Technical Debt 3

2.2 Container Virtualization 3

2.3 SATD in Docker 4

3 Methodology 5

3.1 Dataset 5

3.1.1 Selection of Target Projects 5

3.1.2 Merging Comments 7

3.1.3 Removal of Unnecessary Comments 7

3.2 Manual Classification 7

4 Study Results 10

4.1 RQ1: How many SATDs are present in Docker? 10

4.2 RQ2: What types of SATD exist in Dockerfiles? 11

4.3 RQ3: What is the percentage of each SATD? 16

5 Discussion 18

5.1 Comparison of SATD in general programming language and SATD in Docker 18

5.2 Implications 20

6 Related Work 22

6.1 Software Engineering in SATD 22

6.2 Software Engineering in Docker 23

6.3 Software Engineering in Infrastructure as Code 24

7 Threats to Validity 26

7.1 Internal Validity 26

7.2 External Validity 26

8 Conclusion 28

Acknowledgements 29

References 30

List of Figures

1 Overview of investigation methodology 6
2 Proportion of SATD in manually classified comments 10
3 Our defined classification tree for Docker SATDs. The number in the circle at the upper right of each node indicates the number of instances. 11
4 Proportion of each SATD class and subclass 16
5 Comparison of Java and Dockerfile comment rates 19

List of Tables

1 Definitions and examples of subclasses 12

1 Introduction

Software developers often apply ad hoc solutions or workarounds to achieve their short-term implementation goals [1]. Such solutions are called *technical debt* as a metaphor of "not-quite-right code which we postpone making it right" [2]. 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 [3] [4]. Various SATD studies have been conducted on empirical study [5] [6], automatic detection [7], repayment [8] [9], and domain-specific SATDs (i.e., DNN framework) [10].

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 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 [11] [12]. One powerful benefit of using Docker is from the concept of *infrastructure as code (IaC)* [13]. 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 Docker 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 [14] have been applied to other studies of automatic SATD detection [7] and repayment [8]. 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

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

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 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 thesis, 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 author and two faculty members conducted a manual inspection to classify them using pattern-based SATD detection proposed in previous SATD work [14]. 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 405 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.

The remainder of this thesis 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] `https://hub.docker.com/`

2 Preliminaries

2.1 Technical Debt

In 1993, Cunningham introduced a metaphor, technical debt, to describe an ad hoc solution for programming problems [2]. Since this introduction, many researchers have studied technical debt [15] [16] [17].

Potder et al. [14] 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 et al. [5] conducted a replication study of Potder 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. [10] 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 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 [18] [4]. Therefore, a wide variety of studies are still underway to comprehend [6] [19] [20], detect [21] [7] [22], and repay [8] [9] SATDs.

2.2 Container Virtualization

Container virtualization provides a virtual environment called a container. Since Docker, the current de facto standard for container virtualization [23], was released, container virtualization has rapidly been attracting attention [11].

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 [24], 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

Java projects have usually been used as research subjects in previous SATD studies [5] [18]. 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 [10], 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 Figure 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 [25]. It is assumed that SATDs are also added during the development process. 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. As a result, we collected 462 GitHub repositories and 3,149 Dockerfiles from those repositories. Next, 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.[3] 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] `https://highlightjs.org/`

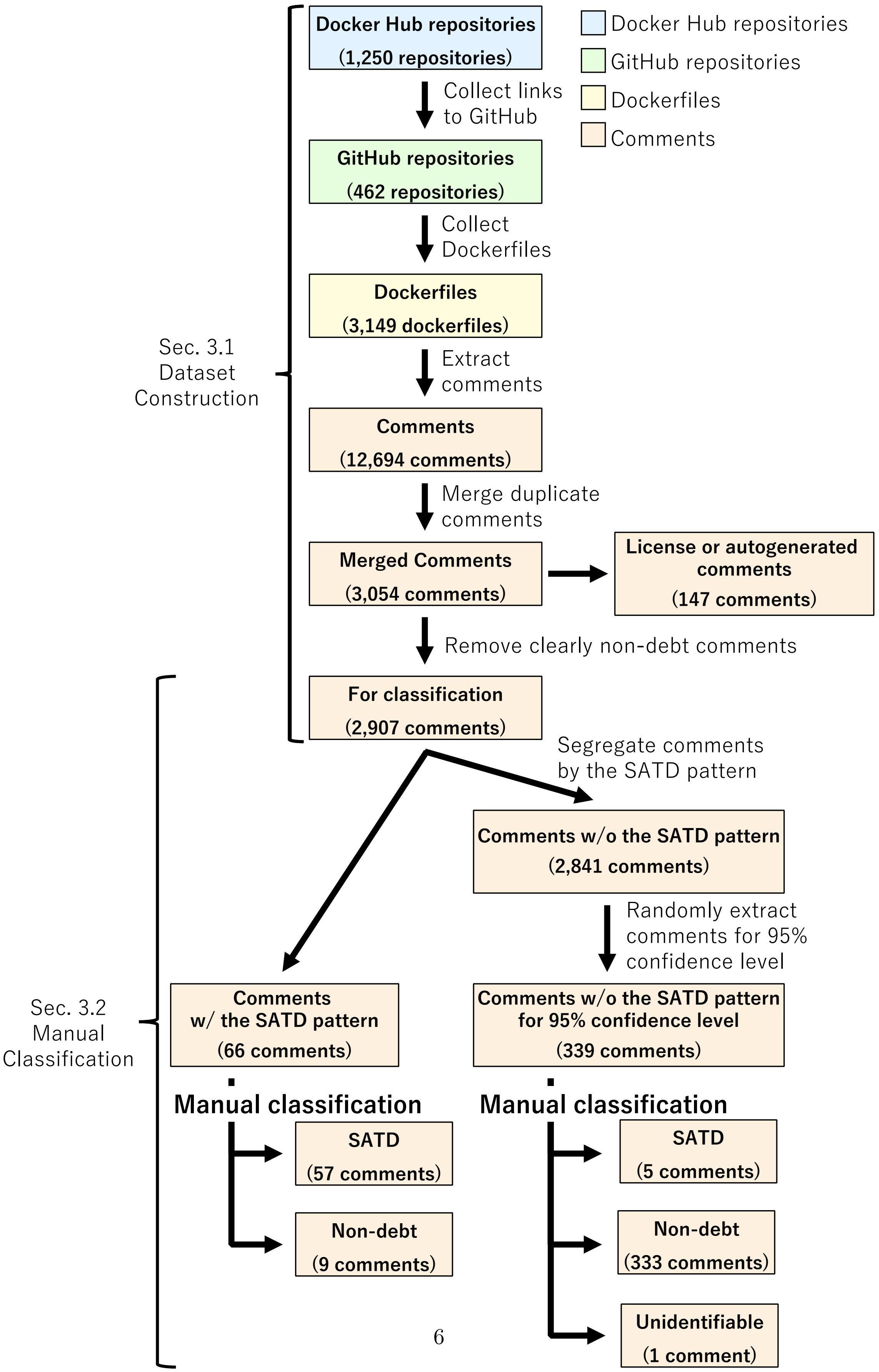
Docker Hub repositories
GitHub repositories
Dockerfiles
Comments
Docker Hub repositories (1,250 repositories)
Collect links to GitHub
GitHub repositories (462 repositories)
Collect Dockerfiles
Dockerfiles (3,149 dockerfiles)
Extract comments
Comments (12,694 comments)
Merge duplicate comments
Merged Comments (3,054 comments)
License or autogenerated comments (147 comments)
Remove clearly non-debt comments
For classification (2,907 comments)
Sec. 3.1 Dataset Construction
Segregate comments by the SATD pattern
Comments w/o the SATD pattern (2,841 comments)
Randomly extract comments for 95% confidence level
Comments w/ the SATD pattern (66 comments)
Comments w/o the SATD pattern for 95% confidence level (339 comments)
Sec. 3.2 Manual Classification
Manual classification
SATD (57 comments)
Non-debt (9 comments)
Manual classification
SATD (5 comments)
Non-debt (333 comments)
Unidentifiable (1 comment)

Figure 1: Overview of investigation methodology

3.1.2 Merging Comments

Many GitHub repositories that handle Dockerfiles have many Dockerfiles in the same repository. The Dockerfiles create images that differ only according to OS, software version, and Linux distribution. For instance, the GitHub repository that manages Dockerfiles of PostgreSQL[4] maintains six different versions of PostgreSQL, each with a different Dockerfile. Moreover, in each version, developers maintain separate Dockerfiles for each of the two Linux distributions (Debian and Alpine). Therefore, in the above case, there are twelve different Dockerfiles in the repository, consisting of six different PostgreSQL versions and two different distributions. [5] The contents of these Dockerfiles are almost the same, which may lead to a bias in the results of the SATD investigation. In fact, many comments extracted from the same repository had the exact same content. Thus, we merged identical comments into a single comment and treated them the same comment. As a result of this aggregation, the total number of comments in the dataset was reduced to 3,054.

3.1.3 Removal of Unnecessary Comments

Dockerfile comments include comments that are clearly not SATD, such as license comments and comments inserted by automatic generation tools (autogenerated comments). Hence, we removed the license and autogenerated comments from the dataset, which were identified because they did not contain the 63 patterns likely to be included in the SATD found by Potder et al. (such as `FIXME` and `hack`, hereinafter, called the "SATD pattern"). [6]) [14]. The non-SATD comments were identified by keywords such as `autogenerated` and `license`. The author then manually confirmed that those comments were, in fact, license or autogenerated comments. As a result, 147 comments were removed, yielding 2,907 comments for classification.

3.2 Manual Classification

In this study, the author and two faculty members conducted a manual classification to investigate how many SATDs exists in Dockerfiles and what types of SATD are present in Docker. In this section, we describe the manual classification process performed by the three classifiers.

[4] `https://github.com/docker-library/postgres`

[5] This is a widely used management practice in Docker that allows the user of the image to select the version and distribution freely.

[6] For more detail, the string `TODO` was added to the 63 patterns. The reason why we added `TODO` is that it is an important keyword for discriminating SATD, although it is not included in the 63 patterns

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 et al. [7] 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 classifier picked comments that they did not understand the reason for the comments' classification. Then, the three classifiers discuss these comments and created clear criteria for each SATD based on the discussion.

Phase 2: Applying SATD classification to Docker domain and defining classification criteria

Before conducting manual classification, 2,907 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 the SATD pattern. In the results, 66 comments included the SATD pattern, and 2,841 comments were determined not to include it. In Phase 2, a total of 100 comments were classified by the three classifiers independently. Of the 100 comments, 50 had the SATD pattern and 50 did not. The classifiers discussed the results of the classification in each phase. Through the discussion, the three classifiers unanimously decided on a single class for each comment. After discussion, it became clear that the SATD classification criteria of Maldonado et al. [7] 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 et al. [5]

Phase 3: Manual Classification of SATD in Dockerfiles

As a result of the classification in Phase 2, about 80% of the comments had the SATD pattern and were considered likely to be SATDs. Therefore, all 66 comments with the SATD pattern were included in the target of the manual classification. In contrast, comments without the SATD pattern, as classified in Phase 2, were considered unlikely to be SATDs. Because their total number was 2,841, it was not easy to classify all of them. Therefore, we manually classified 339 randomly selected comments from this set, which represents a 95% confidence level sample with a 5% confidence interval. In Phase 3, the three classifiers independently and manually classified 16 comments with the SATD pattern and 100 comments without the SATD pattern. The comments classified in Phase 3 were the comments that were not classified in Phase 2. Based on the classification results in Phase 3, the classification criteria were adjusted through discussion.

Phase 4: Reclassification and Completion of Classification

Finally, of the 339 comments without the SATD pattern, 189 comments that had not been classified yet were classified independently by the three classifiers. 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 author visually confirmed all SATDs, and then the three classifiers discussed and reclassified those comments.

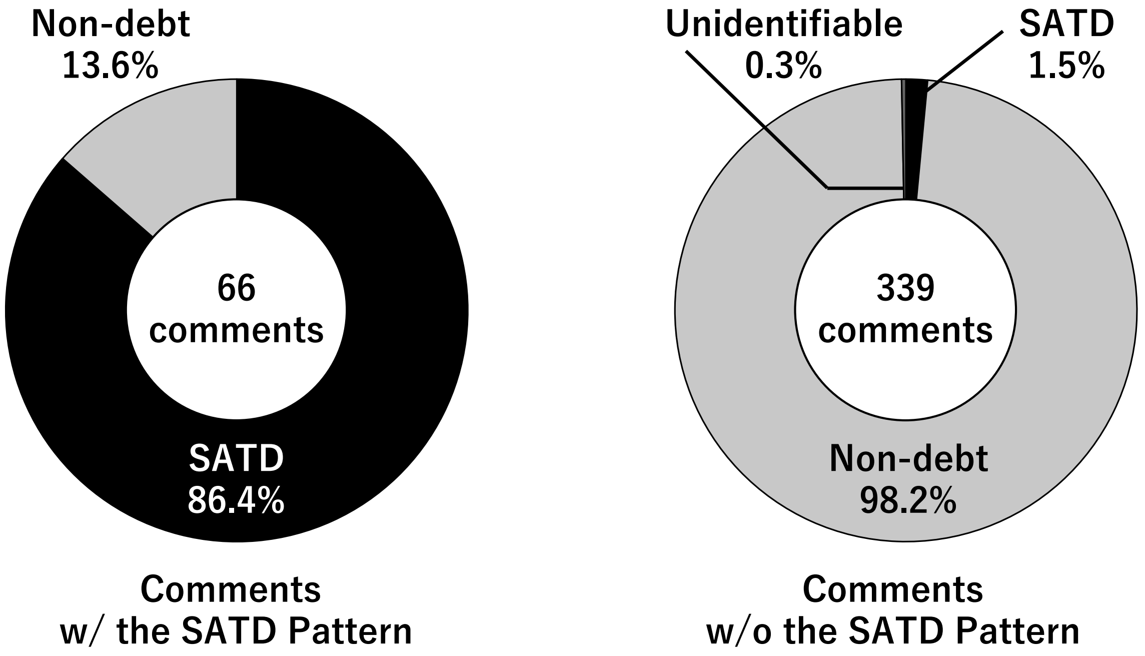

Figure 2: Proportion of SATD in manually classified comments

4 Study Results

4.1 RQ1: How many SATDs are present in Docker?

Figure 2 shows the proportion of SATD in manually classified comments. 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 was inserted its URL in the code. Of the comments that were manually classified, 57 (86.4%) comments with the SATD pattern were SATDs. In contrast, five (1.5%) comments without the SATD pattern were SATDs. If the 2,841 comments without the SATD pattern included SATDs in a similar proportion, it is considered that about 42 of these comments are SATDs. Thus, there are about 99 SATDs in the total of 2,907 comments for classification, which is about 3.4%.

The results of RQ1 yielded the following a finding.

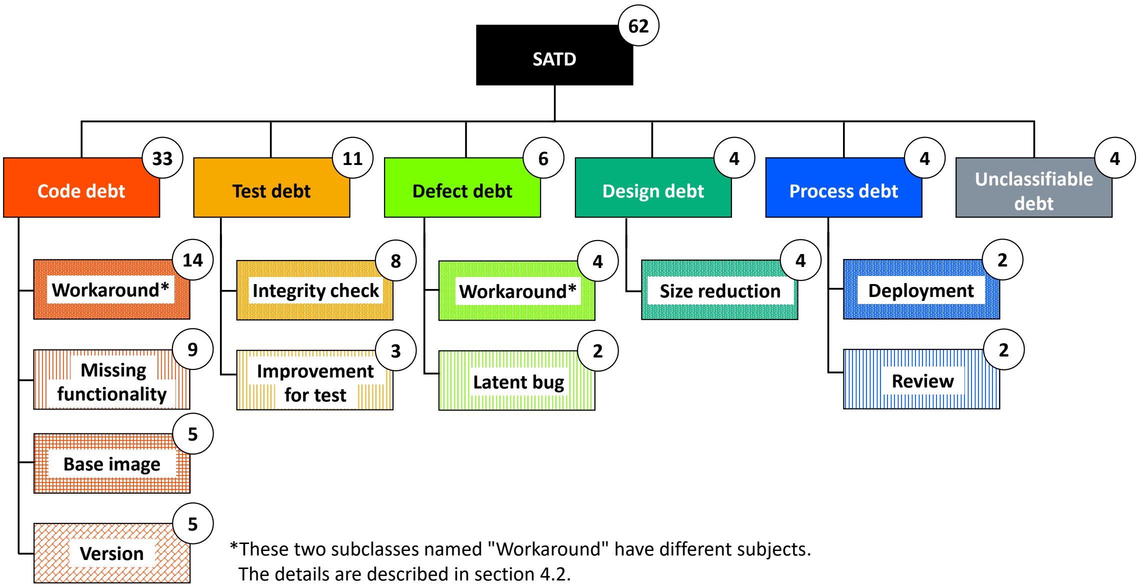

Figure 3: Our defined classification tree for Docker SATDs. The number in the circle at the upper right of each node indicates the number of instances.

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

4.2 RQ2: What types of SATD exist in Dockerfiles?

In the manual classification, 62 SATDs were classified into 5 classes and 11 subclasses. These classification criteria were created by mutual agreement of the three classifiers who conducted the manual classification based on the classification of Bavota et al. [5] Figure 3 shows the classification tree and the results of our manual classification. The six top nodes below the primary "SATD" node in the Figure 3 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 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. [7] The details of these subclasses are described below.

[7] The original text and more information about each example are included in our dataset available at `https://docs.google.com/spreadsheets/d/1ZCkdLxQjJyZpp88NtXYcSCNko8HX-2-uUzX217pf67s/edit?usp=sharing`.

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/Missing functionality SATDs on the lack of functionality inside containers `# TODO: Run both pip and pip3 via virtualenvs`	1777
Code/Base image 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 `# TODO: Pin versions for (url).`	1613
Test/Integrity check SATDs on integrity check `# TODO: Add PGP checking when the feature will be added to build system`	623
Test/Improvement for test 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 `# FIXME: Virtualenv 16.3.0 breaks build, force to use 16.2.0 until fixed.`	737
Defect/Latent bug SATDs on latent bugs `# In PHP 7.4+, the pecl/pear installers are officially deprecated and will be removed in PHP 8+`	538

Design/Size reduction 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
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 `# TODO: aufs-tools`	281

Code debt: Code debt reduces the maintainability of source code. In this study, Code debt is classified into four subclasses: Workaround (14 instances), Missing functionality (9 instances), Base image (5 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/Missing functionality 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/Base image 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 thesis, Test debts are classified into two subclasses: Integrity check (8 instances) and Improvement for test (3 instances).

Test/Integrity check 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/Improvement for test comments ask for improvements in testing methods. The improvements are defined not as fixing bugs in tests but as improving test efficiency and 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 thesis, 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/Latent bug 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 et al. [5] in Java. All the Design debts found in this study are classified as Size reduction (4 instances).

Design/Size reduction 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 [26]. 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 thesis, 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 classifiers 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.

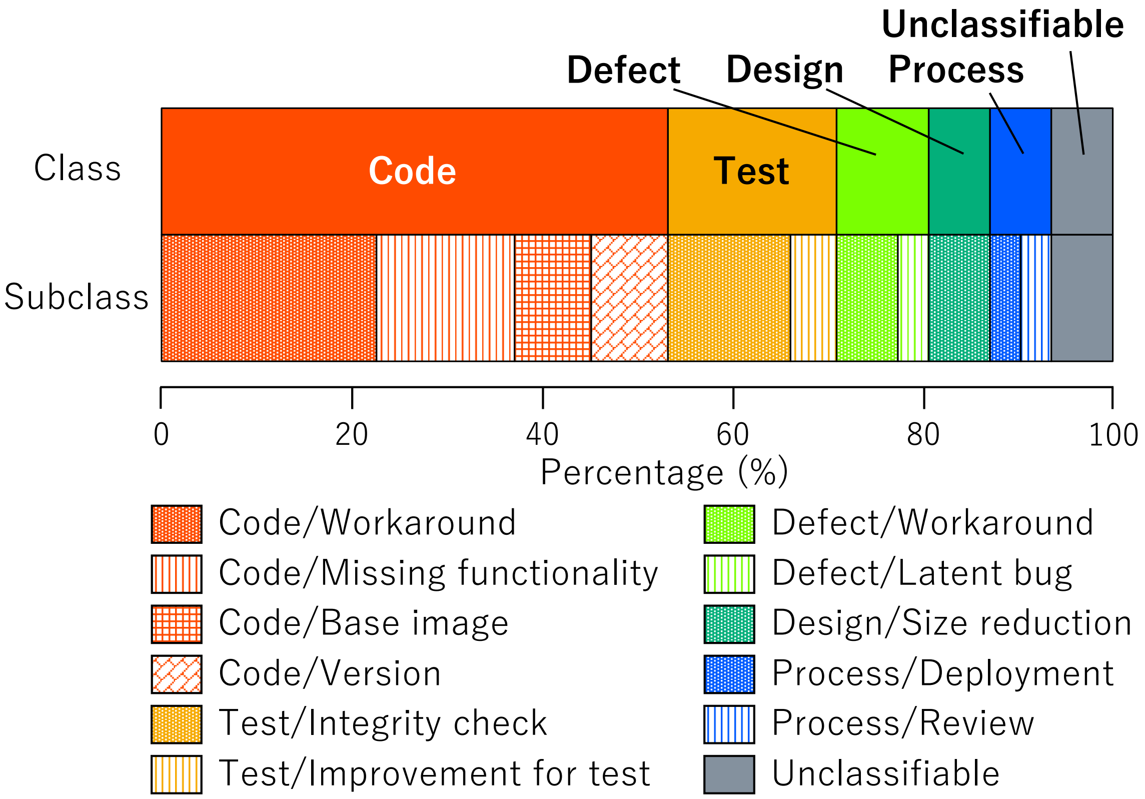

Figure 4: Proportion of each SATD class and subclass

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 4 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 66, 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 Figure 3.

The results show that Code debt is the most common class, accounting for 53.2% of all SATDs, followed by Test debt at 17.7%. The results of SATD classification for Java by Maldonado et al. [7] 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.

The most common subclass is Code/Workaround, which accounts for 22.5% of all

SATDs, followed by Code/Missing functionality and Test/Integrity check. 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 37.1%, which means that about 1/3 of the SATDs are Docker-specific SATDs. The Docker-specific subclasses are Code/Base image, Code/Version, Test/Integrity, Defect/Latent bug, Design/Size reduction, Process/Deployment, and Process/Review.

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 1/3 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 et al. [7]. We show implications for researchers, developers of Docker, and developers of Docker images.

5.1 Comparison of SATD in general programming language and SATD in Docker

As mentioned in Section 4.3, SATD in Docker is considered to have a higher Test debt proportion than SATD in Java. 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.

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 [27]. 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 in the code written by the developers themselves. 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/Latent bugs exist in Dockerfiles because it is susceptible to external factors that are beyond the control of the image developers.

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 frame-

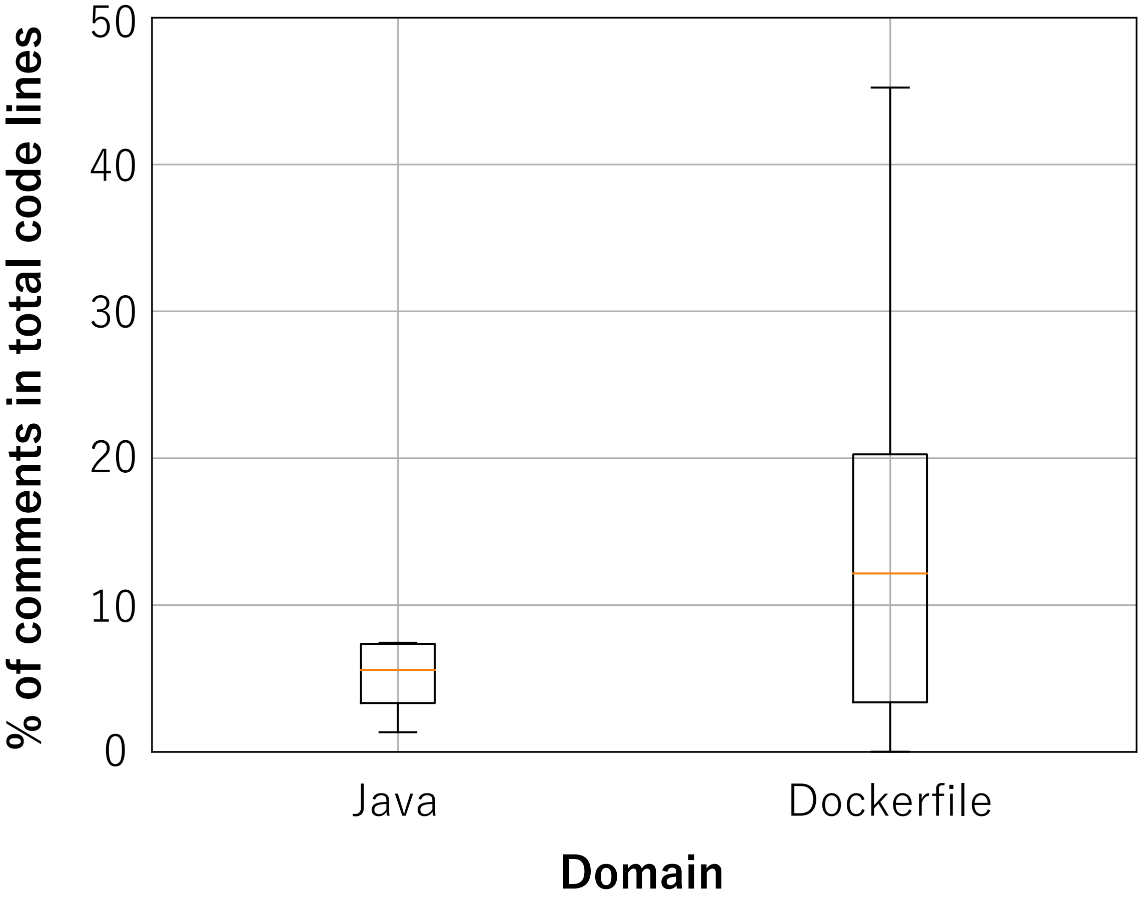

Figure 5: Comparison of Java and Dockerfile comment rates

works. 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 thesis, 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 Maldonado et al. study [7] with the comment rate per GitHub repository used in this thesis. Figure 5 shows boxplots of the result. In this figure, the horizontal axis represents each domain, and the vertical axis represents the comment rate. The orange lines in each boxplot represent medians of them. 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.

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 66 comments with the SATD pattern, there are 57 true positives and 9 false positives. Based on the assumption mentioned in section 4.1, there are approximately 42 SATDs in the comments without the SATD pattern. Therefore, there are 42 false negatives. Then, the remaining 2,799 comments are true 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 86.4% and 57.6%, respectively, yielding an F1 measure of 0.69. Thus, we can use the SATD pattern to identify SATD in Dockerfiles with high accuracy.

5.2 Implications

First, we discuss the implications for Docker researchers. Our analysis shows that there are many SATDs in Dockerfiles, which may increase the importance of research on debt detection and repayment. For example, we can track the SATDs found in the study and investigate how long it takes for the SATDs to be repaid in the future. Furthermore, we might provide a technique of automated SATD detection using machine learning by collecting more SATDs from a larger number of Docker repositories.

In addition, we are making our dataset available to the research community. Our dataset consists of 2,907 comments, including 405 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 5.1, 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. 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. [28] 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. [29] 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, Potder et al. [14] 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 et al. [6] 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 et al. [5] 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. [1] 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 classifiers is the primary method of classifying SATD. Although some studies have classified SATD using automatic detection, such as natural language processing [7], 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 [5] [6] [21]. For example, in the study of SATD classification by Maldonado et al. [6], they investigated ten Java open-source projects. Meanwhile, Liu et al. [10] 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 [26]. 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 [26], 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 [12]. It is also a trendy research area, with more than 7,770 research papers published on Docker since the beginning of 2020. [8]

In software engineering, a wide variety of Docker studies have investigated specific aspects, including the ecosystem of Docker containers [11], Docker build failure [30], and other Docker topics [31] [32]. These studies aimed to examine practical findings and lessons learned by mining a large number of published Dockerfiles. For example, Zhang et al. [33] mined Dockerfiles from 2,840 projects and investigated the evolutionary trajectories of the Dockerfiles. They provided researchers and developers with the following findings: we

[8] `https://scholar.google.com/scholar`(accessed 2020-10-18)

should use official Docker images and reduce the number of image layers to improve image quality. Cito et al. [11] 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. [34] 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. [35] 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 [36] [37], 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 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 [38], Chef [39], and Ansible [40]. Various studies on IaC have been conducted [41] [42]. For example, Sharma et al. [43] 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. [44] 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

In this section, we discuss the possible threats to the validity of our study.

7.1 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 author visually checked them after all classifications were completed. For the SATDs that were considered to need reclassification, the three classifiers discussed and reclassified them in phases. However, because the classification was conducted manually, our subjectivity may have influenced the classification.

In our study, we classified 62 SATDs, obtained through the manual classification of 405 comments. As a result, five classes and eleven subclasses were obtained. Because our classification system resulted from only 62 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.

7.2 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 [34]. 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 [27] 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 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.

Acknowledgements

During this work, I have been fortunate to have received assistance from many people. This work could not have been possible without their valuable contributions. First, I wish to express my deepest gratitude to my supervisor, Professor Shinji Kusumoto at Osaka University, for his continuous encouragement and guidance. I also thank him for providing me an opportunity to do this study.

I would like to express my sincere gratitude to Yoshiki Higo, Associate Professor at Osaka University, for his encouragement and precise advice on how to improve my paper.

I am also profoundly grateful to Shinsuke Matsumoto, Assistant Professor at Osaka University, for his practical discussion and careful direction throughout this study. I learned a lot from him, including his attitude and point of view about research. If it were not for his support, I would never complete this study.

I feel grateful to Yasutaka Kamei, Associate Professor at Kyushu University, for his valuable suggestions and discussion. His assistance in discussions and writing papers helped me to proceed smoothly.

I thank greatly, Tetsushi Kuma, Yuya Tomida, Tasuku Nakagawa, and Kaisei Hanayama, second-year master's students in Kusumoto Laboratory for their constant encouragements and valuable advice.

I would like to thank, Naoto Ichikawa, Ryoko Izuta, Sho Ogino, Akira Fujimoto, and Aoi Maejima, first-year master's students in Kusumoto Laboratory for enriching my research life.

I also thank, Masashi Iriyama, Kanta Kotou, Masayuki Taniguchi, Tomoaki Tsuru, Hiroto Watanabe, fourth-year undergraduate student in Kusumoto Laboratory for their assistance to maintain a good laboratory environment.

I would like to express my deep gratitude to Tomoko Kamiya, a clerical assistant in Kusumoto Laboratory, for her support in many ways.

Finally, I would like to express my utmost gratitude to my family for their continuous encouragement and assistance. They allowed me to study at this university, and I was able to complete my study in the best environment.

References

[1] N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Sp 鱈 nola. Towards an Ontology of Terms on Technical Debt. In *Proc. International Workshop on Managing Technical Debt*, pages 1–7, 2014.

[2] W. Cunningham. The WyCash Portfolio Management System. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1992.

[3] F. A. Fontana, V. Ferme, and S. Spinelli. Investigating the Impact of Code Smells Debt on Quality Code Evaluation. In *Proc. International Workshop on Managing Technical Debt*, pages 15–22, 2012.

[4] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman. Investigating the Impact of Design Debt on Software Quality. In *Proc. Workshop on Managing Technical Debt*, pages 17–23, 2011.

[5] G. Bavota and B. Russo. A Large-Scale Empirical Study on Self-Admitted Technical Debt. In *Proc. IEEE/ACM Working Conference on Mining Software Repositories*, pages 315–326, 2016.

[6] E. D. S. Maldonado and E. Shihab. Detecting and Quantifying Different Types of Self-Admitted Technical Debt. In *Proc. International Workshop on Managing Technical Debt*, pages 9–15, 2015.

[7] E. D. S. Maldonado, E. Shihab, and N. Tsantalis. Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062, 2017.

[8] E. D. S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik. An Empirical Study on the Removal of Self-Admitted Technical Debt. In *Proc. IEEE International Conference on Software Maintenance and Evolution*, pages 238–248, 2017.

[9] S. Mensah, J. Keung, J. Svajlenko, K. E. Bennin, and Q. Mi. On the Value of a Prioritization Scheme for Resolving Self-Admitted Technical Debt. *Journal of Systems and Software*, 135(C):37–54, 2018.

[10] J. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li. Is Using Deep Learning Frameworks Free? Characterizing Technical Debt in Deep Learning Frameworks. In *Proc. International Conference on Software Engineering*, pages 1–10, 2020.

[11] 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 *Proc. International Conference on Mining Software Repositories*, pages 323–333, 2017.

[12] Portworx. Annual Container Adoption Report, accessed 2020-10-05. `https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf`.

[13] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 1st edition, 2010.

[14] A. Potdar and E. Shihab. An Exploratory Study on Self-Admitted Technical Debt. In *Proc. IEEE International Conference on Software Maintenance and Evolution*, pages 91–100, 2014.

[15] 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 *Proc. FSE/SDP Workshop on Future of Software Engineering Research*, pages 47–52, 2010.

[16] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software*, 29(6):18–21, 2012.

[17] E. Tom, A. Aurum, and R. Vidgen. An Exploration of Technical Debt. *Journal of Systems and Software*, 86(6):1498–1516, 2013.

[18] S. Wehaibi, E. Shihab, and L. Guerrouj. Examining the Impact of Self-Admitted Technical Debt on Software Quality. In *Proc. International Conference on Software Analysis, Evolution, and Reengineering*, pages 179–188, 2016.

[19] F. Zampetti, A. Serebrenik, and M. Di Penta. Was Self-Admitted Technical Debt Removal a Real Removal? An In-Depth Perspective. In *Proc. International Conference on Mining Software Repositories*, pages 526–536, 2018.

[20] Y. Kamei, E. Maldonado, E. Shihab, and N. Ubayashi. Using analytics to quantify the interest of self-admitted technical debt. volume 1771, pages 68–71, 2016.

[21] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li. Identifying Self-Admitted Technical Debt in Open Source Projects using Text Mining. *Empirical Software Engineering*, 23(1):418–451, 2017.

[22] M. A. de Freitas Farias, M. G. de Mendonça Neto, A. B. d. Silva, and R. O. Spínola. A Contextualized Vocabulary Model for identifying technical debt on code comments. In *Proc. International Workshop on Managing Technical Debt*, pages 25–32, 2015.

[23] Open Container Initiative. Why have all of these companies come together?, accessed 2020-09-08. `https://opencontainers.org/faq/#why-have-all-of-these-companies-come-together`.

[24] Docker. Docker Overview, accessed 2020-10-28. `https://docs.docker.com/get-started/overview/`.

[25] B. Fluri, M. Wursch, and H. C. Gall. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes. In *Proc. Working Conference on Reverse Engineering*, pages 70–79, 2007.

[26] Docker Documentation. Best Practices for Writing Dockerfiles, accessed 2020-09-17. `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/`.

[27] L. Martinelli. Haskell Dockerfile Linter, accessed 2021-01-18. `https://github.com/hadolint/hadolint`.

[28] N. Zazworka, A. Vetro', C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull. Comparing Four Approaches for Technical Debt Identification. *Software Quality Journal*, 22(3):403–426, 2014.

[29] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda. Towards a prioritization of code debt: A code smell intensity index. In *Proc. International Workshop on Managing Technical Debt*, pages 16–24, 2015.

[30] W. Yiwen, Z. Yang, W. Tao, and W. Huaimin. An Empirical Study of Build Failures in the Docker Context. In *Proc. International Conference on Mining Software Repositories*, pages 76–80, 2020.

[31] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps. A Dataset of Dockerfiles. In *Proc. International Conference on Mining Software Repositories*, pages 528–532, 2020.

[32] J. Henkel, C. Bird, S. K. Lahiri, and T. Reps. Learning from, Understanding, and Supporting DevOps Artifacts for Docker. In *Proc. International Conference on Software Engineering*, pages 38–49, 2020.

[33] Y. Zhang, G. Yin, T. Wang, Y. Yu, and H. Wang. An Insight Into the Impact of Dockerfile Evolutionary Trajectories on Quality and Latency. In *Proc. Annual Computer Software and Applications Conference*, pages 138–143, 2018.

[34] R. Shu, X. Gu, and W. Enck. A Study of Security Vulnerabilities on Docker Hub. In *Proc. Conference on Data and Application Security and Privacy*, pages 269–280, 2017.

[35] A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona. On the Relation between Outdated Docker Containers, Severity Vulnerabilities, and Bugs. In *Proc.*

International Conference on Software Analysis, Evolution and Reengineering, pages 491–501, 2019.

[36] M. U. Haque, L. H. Iwaya, and M. A. Babar. Challenges in Docker Development: A Large-scale Study Using Stack Overflow. In *Proc. International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11, 2020.

[37] M. A. Oumaziz, J. Falleri, X. Blanc, T. F. Bissyandè, and J. Klein. Handling Duplicates in Dockerfiles Families: Learning from Experts. In *Proc. IEEE International Conference on Software Maintenance and Evolution*, pages 524–535, 2019.

[38] S. Krum, W. V. Hevelingen, B. Kero, J. Turnbull, and J. McCune. *Pro Puppet*. Apress, 1st edition, 2014.

[39] M. Taylor and S. Vargo. *Learning Chef: A Guide to Configuration Management and Automation*. O'Reilly Media, Inc., 1st edition, 2014.

[40] Red Hat. Ansible, accessed 2021-01-25. `https://www.ansible.com/`.

[41] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam. Testing Idempotence for Infrastructure as Code. In *Proc. International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 368–388, 2013.

[42] Y. Jiang and B. Adams. Co-evolution of Infrastructure and Source Code - An Empirical Study. In *Proc. Working Conference on Mining Software Repositories*, pages 45–55, 2015.

[43] T. Sharma, M. Fragkoulis, and D. Spinellis. Does Your Configuration Code Smell? In *Proc. Working Conference on Mining Software Repositories (MSR)*, pages 189–200, 2016.

[44] A. Rahman, C. Parnin, and L. Williams. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In *Proc. International Conference on Software Engineering*, pages 164–175, 2019.