# Understanding Self-Admitted Technical Debt in Test Code: An Empirical Study

IBUKI NAKAMURA, Nara Institute of Science and Technology, Japan
YUTARO KASHIWA, Nara Institute of Science and Technology, Japan
BIN LIN, Hangzhou Dianzi University, China
HAJIMU IIDA, Nara Institute of Science and Technology, Japan

Developers often opt for easier but non-optimal implementation to meet deadlines or create rapid prototypes, leading to additional effort known as technical debt to improve the code later. Oftentimes, developers explicitly document the technical debt in code comments, referred to as Self-Admitted Technical Debt (SATD). Numerous researchers have investigated the impact of SATD on different aspects of software quality and development processes. However, most of these studies focus on SATD in production code, often overlooking SATD in the test code or assuming that it shares similar characteristics with SATD in production code. In fact, a significant amount of SATD is also present in the test code, with many instances not fitting into existing categories for the production code. This study aims to fill this gap and disclose the nature of SATD in the test code by examining its distribution and types. Moreover, the relation between its presence and test quality is also analyzed.

Our empirical study, involving 17,766 SATD comments (14,987 from production code, 2,779 from test code) collected from 50 repositories, demonstrates that while SATD widely exists in test code, it is not directly associated with test smells. Our study also presents comprehensive categories of SATD types in the test code, and machine learning models are developed to automatically classify SATD comments based on their types for easier management. Our results show that the CodeBERT-based model outperforms other machine learning models in terms of recall and F1-score. However, the performance varies on different types of SATD.

CCS Concepts: • **Software and its engineering** → *Software design techniques*; *Documentation*; **Software evolution**; **Maintaining software**; **Empirical software validation**; *Software reliability*.

Additional Key Words and Phrases: Self-Admitted Technical Debt, Test Code, Software Quality

**ACM Reference Format:**
Ibuki Nakamura, Yutaro Kashiwa, Bin Lin, and Hajimu Iida. 2025. Understanding Self-Admitted Technical Debt in Test Code: An Empirical Study. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (January 2025), 36 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

Developers often opt for a simpler, albeit non-optimal, implementation over an ideal but time-consuming one due to various reasons such as meeting deadlines or creating rapid prototypes. These implementation choices may introduce additional efforts for future improvement, known as technical debt [16]. While technical debt can accelerate development in the short term, it becomes a

Authors' Contact Information: Ibuki Nakamura, nakamura.ibuki.nh4@naist.ac.jp, Nara Institute of Science and Technology, Ikoma, Japan; Yutaro Kashiwa, Nara Institute of Science and Technology, Ikoma, Japan, yutaro.kashiwa@is.naist.jp; Bin Lin, Hangzhou Dianzi University, Hangzhou, China, b.lin@hdu.edu.cn; Hajimu Iida, Nara Institute of Science and Technology, Ikoma, Japan, iida@itc.naist.jp.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1557-7392/2025/1-ART
https://doi.org/XXXXXXX.XXXXXXX

long-term obstacle, degrading software quality and impeding project progress [39, 77]. As technical debt accumulates, maintenance costs also increase, making it challenging to repay the debt [48, 50].

In practice, developers frequently use keywords like `TODO:` or `FIXME:` in code comments to highlight issues or tasks that need to be addressed. This specific type of technical debt, intentionally introduced by developers, is known as Self-Admitted Technical Debt (SATD) [53]. SATD provides opportunities for researchers and developers to identify technical debt and understand how it is handled in software projects. Given the impact of technical debt on the development process and product quality, researchers have done extensive work to analyze their usage in source code. Vassallo *et al.* [73] found that 88% of developers working on financial systems have introduced SATD. In recent years, it has been reported that SATD impacts software quality [63]. Wehaibi *et al.* [74] compared files with and without SATD, showing that files with SATD tend to have more defects. Both Wehaibi *et al.* [74] and Kamei *et al.* [35] observed that changes related to SATD are more complex than non-SATD changes. Additionally, some studies have manually inspected and classified SATD to demystify it [7, 20]. Farias *et al.* [20] created nine categories of SATD based on technical debt definitions established by the previous studies [1, 2] and detected SATD using a pattern-based approach. Furthermore, Bavota *et al.* [7] conducted a large empirical study and classified SATD into six categories. Based on these categories, Sala *et al.* [58] developed DebtHunter, an ML-based tool to categorize SATD types, further facilitating SATD studies.[1]

However, these studies often overlook or are less eager to investigate SATD in test code. This oversight hinders test-specific issues highlighted by SATDs. For example, a comment such as, "TODO: This test only covers the happy path - need to add edge cases for null inputs," reveals that while the test may pass, it provides incomplete coverage and a false sense of security, leaving the system vulnerable. Furthermore, this gap is evident in the literature; for instance, Bavota *et al.*'s categories [7] primarily use SATD from production code, resulting in no subcategories in their "Test" category, while other categories have many subcategories. Sala *et al.*'s tool [58] also automatically excludes test code when detecting SATD.

Despite this oversight in SATD research, issues in test code have garnered attention from many researchers. For example, van Deursen *et al.* [71] introduced the concept of test smells, which represent issues in test code. These test smells indicate inadequate design or implementation, leading to maintenance difficulties and an increased defect injection rate. Additionally, Li *et al.* [42] conducted a literature review and analyzed previous studies on technical debt (non-self-admitted) and its management. Their results present several subcategories of test-related technical debt. However, the types of listed test-related technical debt are rather limited, and it remains unclear whether the types of SATD would be different compared to general technical debt in the test code.

Studying SATD in test code can help us gain comprehensive insight on how developers document test code issues. Identifying the types of SATDs in test code can help developers better manage the SATD. In this study, we aim to analyze the prevalence and types of SATD in test code and classify their types. The main contributions of our study are as follows.

- We demonstrated the prevalence of SATD in test code and compared it to that in production code. Our result indicates that much fewer instances of SATD reside in the test code, but they are not negligible and still play an important role.
- We examined the relationship between Test SATDs and software quality at the method level. The results reveal that test methods with SATD tend to be more lengthy and prone to code smells. However, no relationship can be found between Test SATD and test smells, implying that these issues occur independently.

[1] https://github.com/PandaMinore/DebtHunter-Tool

- We conducted a manual classification of 506 randomly selected Test SATD instances and identified 20 types of issues, which could be further categorized into 5 main groups. The most common category was "Indicates incomplete or unimplemented tests", highlighting the lack of proper tests in general.
- We built a CodeBERT-based model to automatically classify the types of Test SATD, which outperforms other machine learning models in terms of recall and F1-score, reaching 0.69 and 0.70, respectively. However, the performance varies on different types of SATD.

**Replication Package:** To facilitate replication and further studies, we provide the data and scripts used in our replication package on GitHub.[2]

**Structure of this paper:** Section 2 introduces related work. Section 3 presents motivating examples and the research questions we aim to address. Section 4 describes the methodology adopted to answer the research questions. Section 5 presents the empirical results for each research question. Section 6 summarizes our findings, lessons learned, and describes the implications. Section 7 discusses the threats to validity and Section 8 concludes this paper.

## 2 Related work

In this section, we present related work on self-admitted technical debt, test code issues, and their intersections.

### 2.1 Self-admitted Technical Debt

Potdar and Shihab [53] investigated technical debt explicitly documented as comments in the source code by developers, known as Self-Admitted Technical Debt (SATD). SATD comments often describe issues or incomplete tasks [18], such as *"TODO: - This method is too complex, lets break it up"*[3] in the "ArgoUml" project and *"TODO no methods yet for getClassname"*[4] in the project "Apache Ant". To understand the role SATD plays in software projects, numerous studies have proposed approaches to identify SATD in software and analyzed its relations with software quality.

*2.1.1 SATD Identification.* Various approaches have been proposed to detect SATD in code comments, which are based on either pattern-matching or machine learning.

As one of the earliest approaches, Potdar and Shihab [53] extracted comments from source code using srcML [14] and identified 62 recurring patterns representing SATD. These patterns are further used to match code comments to detect SATD. Maldonado *et al.* [17] extended this work and developed filtering heuristics to remove the comments which are less likely to contain SATD, such as license comments and commented source code. They also manually classified SATDs into five types. Farias *et al.* [19] proposed a Contextualized Vocabulary Model to detect SATD, which systematically takes into account how terms may be combined to identify different types of debt. Guo *et al.* [30] proposed MAT (Matches Task Annotation Tags), an approach for detecting SATD based on tags such as `TODO` and `FIXME`.

Machine learning has also been adopted to detect SATD in recent years. For example, Maldonado *et al.* [18] proposed an NLP-based method, which trains a maximum entropy classifier with a classified dataset to automatically detect SATDs related to design and requirements. Liu *et al.* [43] developed SATD Detector, which is based on a composite classifier of several sub-classifiers using Naive Bayes Multinomial (NBM). Their tool provides an environment for developers to manage SATD comments through an Eclipse plug-in. Ren *et al.* [56] proposed a method to detect SATD

[2] https://github.com/ibu00024/UnderstandingSelfAdmittedTechnicalDebtinTestCode
[3] https://github.com/argouml-tigris-org/argouml/blob/VERSION_0_34/src/argouml-app/src/org/argouml/notation/providers/uml/AbstractMessageNotationUml.java#L448
[4] https://github.com/apache/ant/blob/rel/1.7.0/src/main/org/apache/tools/ant/util/ClasspathUtils.java#L496

using convolutional neural networks (CNN). Sala *et al.* [58] developed "DebtHunter," a tool that first identifies SATD and then classifies SATD into specific debt types. Both classifiers used in these two steps are built on Sequential Minimal Optimization (SMO).

*2.1.2 Prevalence of technical debt.* Potdar and Shihab [53] analyzed the quantity and reasons for introducing SATD in four large-scale open-source software repositories. They found that SATD was present in 2.4% to 31% of the files in the projects, with only 26.3% to 63.5% of the issues being resolved after introduction. They also discovered that SATD is often introduced by experienced developers and tends to be added regardless of the release timing. Maldonado and Shihab [17] investigated SATD in five open-source software repositories and classified SATD into five categories as indicated by Alves *et al.* [3]. Among these types, Design Debt, which indicates problems that violate design principles, accounts for the largest proportion of cases (ranging from 42% to 84%), followed by Requirement Debt, which indicates problems related to requirements (ranging from 5% to 45%). Also, Vassallo *et al.* [73] conducted a survey on testing activities and the management of technical debt in financial systems. They found that many developers documented technical debt in a self-reported manner, with 88% of developers implementing SATD.

*2.1.3 Relation between SATD and software quality.* Wehaibi *et al.* [74] compared the number of defects in files with and without SATDs in five OSS projects and found no clear difference. However, they observed that the number of defects tended to increase after SATDs were introduced. Additionally, while future defects were less likely to occur after changes involving SATD, these changes were usually more complex than changes to files without SATD. Based on this finding, they concluded that the existence of SATD makes system changes more difficult. Kamei *et al.* [35] investigated the complexity and dependency of code changed during the introduction and removal of SATD to measure the amount of "interest" it accumulated. Based on the Apache JMeter project, they found that 42% to 44% of the code incurred positive interest, meaning it would cost more to remove in the future. Conversely, 8% to 13% incurred negative interest, and 42% to 49% did not incur interest.

Palomba *et al.* [51] studied the relationship between SATD and refactoring in three OSS projects. They found that approximately 46% of refactored classes had SATD in previous versions. Furthermore, in 67% of cases, comments and descriptions related to SATD were deleted by refactoring, resolving the technical debt. This study shows that SATD might play an important role as a motivation for refactoring. Maldonado *et al.* [18] analyzed the relationship between SATD and code smells. Code smells indicate potential flaws in the design or implementation of production code [26], which are likely to cause problems in the future. The authors analyzed the overlap between SATD and three representative types of code smells: Long Method, God Class, and Feature Envy at the file level. Their investigation, which targeted 10 OSS repositories, revealed that 65% of files containing SATD also had Long Methods, 44.2% contained God Classes, and 20.7% contained Feature Envy classes. Moreover, when considering all code smells, they found that an average of 69.7% of files containing SATD had at least one code smell. From these findings, they concluded that while there are certain overlaps between SATD and code smells, these two indicators can serve as complementary approaches for detecting technical debt.

Rantala *et al.* [54] investigated the relationship between keyword-labeled self-admitted technical debt (KL-SATD) [55], such as "TODO" or "FIXME", and issues identified by static code analysis using SonarQube. Their results indicate that KL-SATD is associated with reduced code maintainability as measured by SonarQube. The introduction and removal of KL-SATD are mainly related to code smells rather than vulnerabilities or bugs. However, there is a limited overlap between KL-SATD and SonarQube issues, with only 36% of KL-SATD comments being in the context of a SonarQube issue, and only 15% directly addressing an issue.

These studies do not distinguish whether the SATD is present in the production code or the test code.

### 2.2 Issues in Test code

Van Deursen *et al.* [71] coined the concept of test smells, representing potential problems in test code, which are often used to investigate their impact on software quality and maintainability [27]. Bavota *et al.* [6] investigated the prevalence of test smells in test code and their effect on software understanding and maintenance. They found that 86% of the test cases for 27 software systems contained at least one test smell, and the presence of test smells reduced the efficiency of program comprehension. Tufano *et al.* [70] conducted an empirical study involving 152 projects to investigate the nature of test smells. Their results revealed that test smells tend to persist in the system for a long time, with 80% not being fixed even after 1,000 days. Spadini *et al.* [65] studied the impact of test smells on the quality of test code. Their analysis of 10 OSS projects revealed that tests with test smells are 47% more likely to be modified and 81% more likely to have defects than code without test smells.

These aforementioned studies have been conducted to clarify test-code-related issues that are not admitted by developers. These studies explore different types of test issues. Studying SATD in test code could provide new insights into what issues developers face in test code and help to identify new types of test smells and (non-self-admitted) technical debt.

### 2.3 Testing-Related SATD

Gat and Heintz [28] investigated the impact of technical debt on software development processes and proposed methods for its reduction. They found that the lack of automated testing is a major factor causing delays in development speed and recommended enhancing unit testing and utilizing "Test as a Service." Codabux and Williams [13] also examined the challenges and best practices in managing technical debt. They identified "Automation debt," caused by a lack of test automation, and "Test debt," arising from unexecuted tests.

Li *et al.* [42] conducted a literature survey on technical debt published between 1992 and 2013. They categorized technical debt into ten different categories, identifying one relevant to testing, known as Test Technical Debt, which is the third most well-studied category. The test category includes seven subcategories: "Low code coverage," "Deferring testing," "Lack of tests," "Lack of test automation," "Residual defects not found in tests," "Expensive tests," and "Estimation errors in test effort planning."

These identified test-related SATD types are either over coarse-grained or not extracted in a systematic manner (*i.e.,* by aggregating multiple independent studies).

## 3 Research Questions

### 3.1 Motivating Examples

Bavota *et al.* [7] conducted an empirical study on SATD using 159 repositories from Eclipse and Apache projects. They manually classified 366 SATDs into six categories: Code, Design, Document, Defect, Test, and Requirement. Code debt accounted for the largest proportion (30%), followed by Defect debt and Requirement debt, each accounting for 20%. Design, Document, and Test debt accounted for 12%, 10%, and 8%, respectively. More importantly, most of the categories have subcategories except Test debt. For example, Code debt has two subcategories: "low internal quality" indicating poor code quality, and "workaround" referring to compromised code due to temporary implementations.

In software projects, we have noticed that SATD comments in the test code do have different intentions:

- **Test failures**
  `// TODO: Passes on macOS, fails on Linux and Windows with AccessDeniedException.`
- **Need for specific tests**
  `// TODO : more tests for datetimes with timezones and/or offsets`
- **Special implementation for testing purposes**
  `// TODO: This is a hack, wc.login does not work with the form`

Some studies have also observed a very small number of instances relevant to testing when they studied SATD mainly in production code. Farias *et al.* [20] proposed a pattern-based method for detecting SATD and conducted an empirical evaluation on three repositories: JEdit, Lucene, and ArgoUML. During the evaluation, they observed three types of Test debt: "deficiencies in testing activities" indicating defects in testing, "tests to do" indicating tests that should be conducted, and "insufficient code coverage" indicating low test coverage. However, these subcategories were derived from a small number of samples (*i.e.,* only 9 instances). Kashiwa *et al.* [36] also reported test-related SATD when investigating the impact of SATD in modern code reviews. They randomly sampled 375 SATD comments from OpenStack and Qt projects and classified them into six types: Scheduling, Work Dependency, Communication, Problem Report, Workaround, and Test. They observed 10 instances in the Test category and identified two subcategories: Necessity and Failure. The former refers to the cases where sufficient tests are missing for the method, while the latter indicates that tests fail at the location containing the SATD. Azuma *et al.* [5] investigated SATD in Docker using Dockerfiles collected from the top 1250 images on Docker Hub. They manually classified 382 comments and found 50 SATD instances, which were further categorized into five types: Code Debt, Test Debt, Defect Debt, Design Debt, and Process Debt. Two subcategories were identified for Test debt: "integrity check" (lack of an integrity check on binary files or hash values used in a container) and "improvement for test" (asking for improvements in testing methods). According to their study, while there were only 9 cases of SATD related to testing, Test Debt was the second most common type of SATD after Code Debt, which had the most cases (26).

While researchers have created several subcategories for test debt, they are normally derived from a small size of samples. Moreover, they are mainly a side product when studying SATD in whole software projects with no specific attention to testing. In this study, we collected 2,779 SATD instances present in the test code from 50 repositories to reveal the prevalence of Test SATD. We also manually classified 506 samples to disclose the intentions behind Test SATD. Studying SATD in test code can help us understand what test-related issues developers recognize and how they are handled. Moreover, we might identify new test smells with Test SATD.

### 3.2 Proposed Research Questions

In this study, we aim to answer the following four research questions (RQs). Note that this study refers to SATD in production code as `Production SATD` and SATD in test code as `Test SATD`.

**$RQ_1$: How prevalent is SATD in test code?**

A previous study by Bavota *et al.* [7] has claimed that test-related SATD is the least frequent one by manually inspecting 366 SATD instances. However, the SATD instances were collected from only two ecosystems (Apache and Eclipse), which hinders the generalizability of the finding. Additionally, the original study was conducted in 2016, and there are currently more advanced SATD detection tools available, which might lead to different results. As the first step of this study,

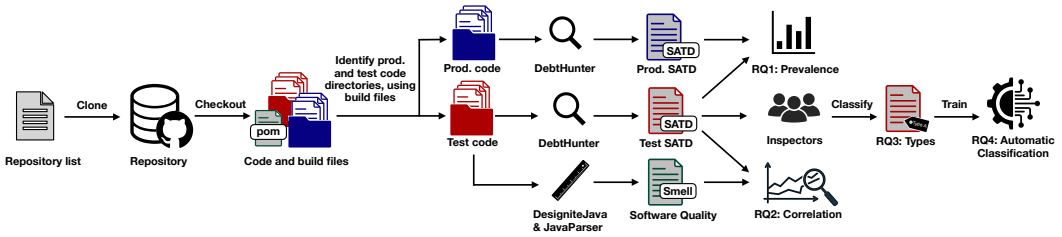



Fig. 1. Overview of the data analysis

we aim to replicate the study of Bavota *et al.* to reveal the prevalence of SATD in test code, collected from a much more diverse repositories.

*RQ*$_2$**: Is SATD in test code correlated with test quality?**

Previous research on SATD has investigated the relationship between SATD and various software quality indicators, such as complexity metrics [53], coupling and readability [7], code smells [18, 54]. These studies suggest that SATD is not directly correlated with many aspects of code quality, although there is certain correlation between SATD and code smells. It is worth noting that these studies mainly target SATD in production code, and there are currently no studies examining the relationship between Test SATD and test quality. In RQ2, we aim to fill this gap.

*RQ*$_3$**: What are the purposes of SATD in test code?**

Existing studies [7, 36] have identified several test-related SATD. However, they are either too coarse-grained or derived from a small size of samples. To get a comprehensive picture of what types of SATD exist in the test code, in RQ3, we aim to identify the Test SATD types in a systematic manner by manually analyzing Test SATD instances and categorizing them based on the intentions of developers.

*RQ*$_4$**: To what extent can SATD in Test Code be classified automatically?**

Various approaches [12, 18, 58] have been proposed for detecting and classifying SATD. However, these tools mainly focus on production code and no tool has been developed to categorize SATD in test code. In RQ4, we aim to fill this gap and develop a classifier to categorize Test SATD based on the categories defined in *RQ*$_3$: What are the purposes of SATD in test code?. We believe that the new model could streamline future analysis of Test SATD.

## 4 Methodology

This section describes the data collection and analysis process in this study.

### 4.1 Data Collection

Figure 1 shows the overview of our data collection process. To collect SATD instances from diverse software repositories, we select 50 projects: 20 projects from previous SATD studies [18, 30] and 30 projects from testing-related studies [9, 37, 47, 64]. This is in line with our goal to study Test SATD as previous SATD studies mainly focus on production code. Table 1 summarizes the repositories used in this study.

For each of these repositories, we clone and checkout the latest revision of the master/main branch. However, for some repositories, we collected specific snapshot versions instead. We then identify SATD in the production code and the test code. Since all 50 repositories in our study are Java projects using either Maven or Gradle, we identified production and test directories by analyzing their build configuration files. We located each repository's `pom.xml` file (Maven projects) or `build.gradle` file (Gradle projects) and parsed it to extract directory paths. For Maven projects,

Table 1. Statistics of the studied repositories

| # | Repository | P.LOC | T.LOC | Files | SATDs | Stars | Commits | Forks | Dev. | Ref |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Ant | 114,501 | 32,279 | 1,327 | 243 | 442 | 14,963 | 445 | 80 | [18] |
| 2 | ArgoUML | 160,769 | 16,705 | 1,908 | 1,828 | 262 | 17,797 | 102 | 5 | [18] |
| 3 | Columba | 66,394 | 1,526 | 993 | 138 | - | - | - | - | [18] |
| 4 | EMF | 606,645 | 105,084 | 3,140 | 141 | 20 | 10,341 | 24 | 36 | [18] |
| 5 | Hibernate | 163,448 | 166,085 | 4,705 | 812 | - | - | - | - | [18] |
| 6 | Jedit | 121,532 | 3,502 | 617 | 164 | - | - | - | - | [18] |
| 7 | JFreeChart | 97,460 | 41,664 | 1,017 | 93 | 1,307 | 3,894 | 565 | 30 | [18] |
| 8 | Jmeter | 120,620 | 28,143 | 1,402 | 272 | 8,921 | 18,254 | 2,193 | 80 | [18] |
| 9 | Jruby | 265,506 | 10,427 | 1,793 | 1,016 | 3,835 | 53,597 | 928 | 362 | [18] |
| 10 | Squirrel | 186,681 | 28,553 | 2,325 | 253 | 75 | 7,766 | 19 | 20 | [18] |
| 11 | Dubbo | 154,678 | 108,908 | 3,581 | 168 | 41,184 | 7,535 | 26,527 | 401 | [30] |
| 12 | Gradle | 494,771 | 17,712 | 10,392 | 698 | 17,886 | 114,478 | 4,984 | 345 | [30] |
| 13 | Groovy | 191,447 | 12,262 | 1,793 | 462 | 5,336 | 20,871 | 1,911 | 359 | [30] |
| 14 | Hive | 867,928 | 178,970 | 5,086 | 1,354 | 5,747 | 10,189 | 4,757 | 257 | [30] |
| 15 | Maven | 55,245 | 19,455 | 1,012 | 173 | 4,716 | 10,966 | 2,764 | 209 | [30] |
| 16 | Poi | 281,995 | 142,878 | 3,710 | 818 | 2,062 | 12,828 | 792 | 17 | [30] |
| 17 | Spring Framework | 383,270 | 391,933 | 8,310 | 283 | 58,540 | 29,550 | 38,600 | 355 | [30] |
| 18 | Storm | 249,986 | 40,390 | 2,145 | 103 | 6,644 | 10,798 | 4,064 | 281 | [30] |
| 19 | Tomcat | 268,488 | 92,832 | 2,616 | 814 | 7,891 | 25,797 | 5,211 | 133 | [30] |
| 20 | Zookeeper | 61,382 | 59,789 | 925 | 109 | 12,549 | 2,534 | 7,295 | 238 | [30] |
| 21 | Commons IO | 17,890 | 33,866 | 513 | 87 | 1,036 | 4,849 | 695 | 115 | [37] |
| 22 | Spring | 2,079 | 4,240 | 123 | 5 | 2,886 | 2,165 | 2,628 | 49 | [37] |
| 23 | Joda-Beans | 17,353 | 34,625 | 328 | 2 | 146 | 891 | 40 | 11 | [37] |
| 24 | Jsoup | 15,306 | 14,745 | 153 | 58 | 11,222 | 1,969 | 2,248 | 103 | [37] |
| 25 | Spark | 6,222 | 5,048 | 184 | 5 | 9,660 | 1,041 | 1,569 | 123 | [37] |
| 26 | LittleProxy | 4,180 | 4,665 | 88 | 3 | 2,095 | 998 | 782 | 27 | [37] |
| 27 | RxJava JDBC | 4,611 | 3,340 | 79 | 7 | 803 | 926 | 115 | 13 | [37] |
| 28 | Spoon | 83,543 | 83,893 | 2,407 | 148 | 1,840 | 4,651 | 361 | 134 | [37] |
| 29 | Accumulo | 441,912 | 44,357 | 2,179 | 110 | 1,104 | 9,001 | 463 | 158 | [47] |
| 30 | BookKeeper | 157,229 | 121,242 | 2,351 | 62 | 1,955 | 3,322 | 960 | 204 | [47] |
| 31 | Camel | 1,041,075 | 653,891 | 21,464 | 1,999 | 5,906 | 70,277 | 5,048 | 331 | [47] |
| 32 | Cassandra | 314,778 | 273,241 | 3,749 | 496 | 9,292 | 28,567 | 3,723 | 272 | [47] |
| 33 | CXF | 389,365 | 308,696 | 7,513 | 443 | 885 | 18,004 | 1,442 | 238 | [47] |
| 34 | Flink | 838,513 | 748,068 | 13,521 | 696 | 25,086 | 34,899 | 13,717 | 285 | [47] |
| 35 | Hadoop | 997,669 | 902,657 | 11,871 | 1,022 | 15,190 | 27,121 | 9,076 | 303 | [47] |
| 36 | Kafka | 271,653 | 356,138 | 4,245 | 327 | 30,571 | 11,728 | 14,524 | 347 | [47] |
| 37 | Karaf | 105,567 | 23,311 | 1,603 | 135 | 691 | 9,471 | 664 | 163 | [47] |
| 38 | Wicket | 142,446 | 77,054 | 3,306 | 141 | 768 | 21,683 | 393 | 116 | [47] |
| 39 | Struts | 123,978 | 87,049 | 2,589 | 179 | 1,318 | 7,160 | 824 | 77 | [9] |
| 40 | Compress | 43,544 | 29,119 | 623 | 170 | 370 | 5,121 | 295 | 95 | [9] |
| 41 | Jenkins | 113,619 | 75,714 | 1,782 | 509 | 24,213 | 34,958 | 9,093 | 296 | [9] |
| 42 | Spring Security | 103,939 | 167,283 | 3,145 | 64 | 9,224 | 15,713 | 6,092 | 388 | [9] |
| 43 | FileUpload | 2,221 | 4,205 | 92 | 13 | 248 | 1,657 | 187 | 49 | [9] |
| 44 | Imaging | 30,710 | 11,347 | 623 | 73 | 461 | 2,486 | 197 | 45 | [9] |
| 45 | Sling | 7,246 | 5,750 | 179 | 8 | 15 | 687 | 22 | 21 | [9] |
| 46 | UAA | 62,053 | 124,466 | 1,646 | 24 | 1,625 | 10,952 | 834 | 158 | [9] |
| 47 | Jackson | 6,283 | 10,702 | 195 | 9 | 600 | 1,560 | 230 | 41 | [9] |
| 48 | Prime JWT | 3,759 | 2,809 | 101 | 4 | 187 | 326 | 43 | 12 | [9] |
| 49 | OpenRefine | 57,094 | 28,418 | 1,026 | 213 | 11,446 | 7,995 | 2,080 | 353 | [9] |
| 50 | Quarkus | 540,637 | 449,513 | 16,729 | 812 | 14,764 | 45,217 | 2,900 | 391 | [64] |

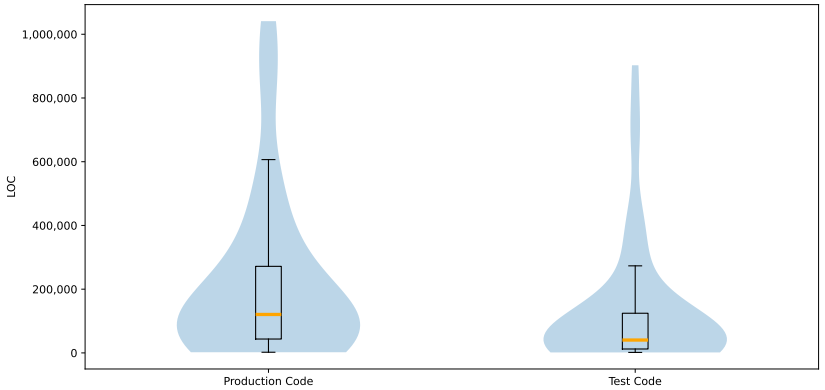



Fig. 2. Distributions of LOC in the production and test Code

we extracted paths from the `sourceDirectory` and `testSourceDirectory` elements; for Gradle projects, we identified paths from the `main` and `test` properties. When these paths were not explicitly defined, we used standard Java conventions: `src/main/java` for production code and `src/test/java` for test code.[5,6] In the end, we identify 100,364 files in the production directories and 60,515 files in the test directories. From these files, we identify SATDs in the production code and test code. We consider SATD instances found in the production code as `Production SATDs` and those found in the test code as `Test SATDs`. For SATD detection, we employ a state-of-the-art SATD detection tool, `DebtHunter` [58]. `DebtHunter` is an ML-based tool that uses the Sequential Minimal Optimization (SMO) algorithm [10]. A previous study [58] has demonstrated the high performance of `DebtHunter`, with a precision of 0.972, recall of 0.967, and F1-score of 0.965, outperforming other SATD detection tools. Additionally, since DebtHunter has been trained with multiple labels for classifying the types of SATD, we also use this tool to identify SATD types. Note that we modified the source code of `DebtHunter` so that it can detect Test SATD because the original version excludes the test code.

## 4.2 Data Analysis

*4.2.1 RQ1 (Prevalence of Test SATD).* To understand how prevalent SATD is in the test code, we count the number of Test SATD and compare it with that of Production SATD. In software projects, the size (*i.e.,* lines of code) of the production code and test code differ significantly. Figure 2 shows the distribution of LOC of the production code and test code. The median numbers of LOC in the production code and test code are 121,076 and 37,507.5, respectively. To ensure a fair comparison, we measure the number of SATD instances per 10,000 lines of code. Finally, we apply the Wilcoxon signed-rank test [75], which is a non-parametric test to compare paired data and confirm a statistically significant difference ($\alpha < 0.01$). We also measure the effect size using Cliff's Delta ($d$) [29]. We follow the well-established guideline [29] to interpret the effect size: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$.

[5] https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html
[6] https://docs.gradle.org/current/userguide/java_plugin.html#sec:java_project_layout

*4.2.2 RQ2 (Relationship with Test Quality) .* To understand how Test SATD is correlated with the test quality, we use several proxies to represent the test quality, including general code quality metrics (code smells, lines of code (LOC), complexity, and readability) and test-specific metrics (test smells, assertions, and annotations) at the method level. The LOC, the number of assertions, and annotations are counted after parsing the source code with `JavaParser`.[7]

The test smells are detected with `tsDetect` [52],[8] a state-of-the-art tool which can detect 16 of 19 test smells proposed by van Deursen *et al.* [71] at the method level. Their evaluation shows an average precision and recall exceeding 95% for detecting each type of smell. To detect code smells, we use `Designite Java`,[9] which can detect 10 of 11 code smells proposed by Sharma *et al.* [61]. To ensure the reliability of these automated detectors on our specific dataset, we conducted an experimental manual validation. We randomly sampled 100 instances identified as test smells by `tsDetect` and 100 instances identified as code smells by `Designite Java`. Two of the authors then independently inspected each instance to verify if it was a true positive. Any disagreements were resolved through discussion with a third inspector to reach a consensus. Our validation confirmed a high precision for both tools: 90% for the test smell detection and 94% for the code smell detection. These results to a high extent guarantee the validity of the quality metrics used in our analysis.

We also use `Designite Java` to calculate the cyclomatic complexity of each method, which is represented as an integer with a minimum value of 1, where a higher value indicates greater complexity. For readability, we use the tool proposed by Scalabrino *et al.* [59]. This tool utilizes a comprehensive model that integrates textual features, such as identifiers and comments, with structural code features. The readability score is evaluated on a scale from 0 to 1, where a higher value indicates better readability. Note that code comments increase the readability score, and SATD comments may increase the score as well. To examine whether SATDs are included in the methods with low readability independent of SATD comment influence, we removed SATD comments in the test methods before applying the tool.

Next, these data are matched with the same method to investigate the relationship between various software qualities and Test SATD. For test smells and code smells, Chi-square test [67] is conducted to examine the relationship between the presence or absence of each smell. The null hypothesis is set as "The presence or absence of Test SATD and the presence or absence of each smell are independent," and tested at the significance level of $\alpha = 0.05$. The presence or absence of Test SATD was determined at the method level, based on whether SATD exists in the documentation comments written for the method or in the comments contained within the method. Furthermore, to investigate whether there is a difference in the number of each smell based on the presence or absence of Test SATD, a non-parametric test, the Mann-Whitney U test [46], is used to calculate the p-value. The null hypothesis is "There is no difference in the number of smells based on the presence or absence of Test SATD," and the test is conducted with a significance level of $\alpha = 0.05$. In addition, we used the U-score obtained during the computation of the Mann-Whitney U test to obtain a Z-score and calculate the effect size. The effect size r [45] is determined using the formula $r = |z|/\sqrt{SampleSize}$ and ranges between 0 and 1. Generally, an effect size $r$ between 0.1 and 0.3 indicates a small effect, values greater than 0.3 and up to 0.5 indicate a medium effect, and values greater than 0.5 indicate a large effect [15]. Similarly, we measure LOC, the number of assertions, annotations, cyclomatic complexity, and readability, when the test method does/does not contain Test SATDs. And then we examine the statistically significant difference between these two groups using the Mann-Whitney U test.

[7] https://github.com/javaparser/javaparser
[8] https://github.com/bin-lin/TestSmellDetector
[9] https://github.com/tushartushar/DesigniteJava

*4.2.3 RQ3 (Purposes of Test SATD).* To understand developers' intentions behind Test SATD, we perform a manual inspection to categorize Test SATD. To facilitate the classification, we apply several filters to exclude comments that do not contain meaningful content from the collected Test SATD in the data collection process. Specifically, we remove comments that contain only tags like `TODO` or `FIXME` without any other text. This filtering eliminates 651 SATD instances from the original 2,779 instances. Additionally, we remove comments that are automatically inserted by integrated development environments (IDEs) such as `TODO Auto-generated method stub`. This step removes 36 SATD instances, leading to 2,092 remaining.

Next, we randomly select 506 SATD comments, which represent a statistically significant sample with a 99% confidence level and a 5% confidence interval. For the classification, we adopted an inductive coding approach, allowing categories to emerge directly from the data rather than starting with pre-existing taxonomies from studies on production code or non-self-admitted technical debt. This methodological decision was made to avoid categorization bias, as pre-existing categories developed from production code might overlook patterns and issues unique to the test code context. By developing the taxonomy from the ground up, we aimed to ensure that it precisely represents how developers document debt in testing contexts and to enable the discovery of test-specific insights.

The classification itself was conducted using a multi-stage, iterative approach based on card sorting [66], consisting of five iterations with approximately 100 instances each. In the first iteration, three authors independently assigned free-form labels to the initial 100 SATD instances to capture their intent, allowing for variations in wording for similar concepts. Afterward, the three authors discussed these labels. By resolving disagreements and grouping semantically similar labels, they created an initial set of categories. In the subsequent four iterations, two authors independently classified the remaining samples (in batches of approximately 100) using this initial category set. A third author solved any disagreements that arose through the discussion with the others. If an instance did not fit any existing category, a new category was proposed and added to the set after a discussion among the three authors. The three annotators involved in this process have 5-15 years of programming experience.

*4.2.4 RQ4 (Automatic classification).* To examine the extent to which we can accurately classify the SATD types, in RQ4, we construct a classifier following the approach by Sabbah *et al.* [57] that classifies production SATD types using natural language processing word embeddings. We utilize the comment texts and labels assigned in RQ3 for the training and testing process.

Prior to training, we pre-process the comments by removing punctuation marks and HTML tags. We then tokenize the comments, remove stop words, and perform stemming. The tokens are converted into numerical vectors using Term Frequency-Inverse Document Frequency (TF-IDF) [34]. We also employ various machine-learning algorithms used in the previous studies [12, 25, 32], including Support Vector Machines (SVM) [72], Naive Bayes (NB) [38], Random Forest [8] and eXtreme Gradient Boosting (XGBoost) [11].

Additionally, we developed language model-based classifiers due to their high classification accuracy demonstrated in previous studies [57]. Specifically, we employed BERT [21] and CodeBERT [24], a pre-trained model capable of understanding the context of source code and natural language. In our study, we replace the output layer of the model with a linear layer to enable the model to classify SATD comments into five sub-categories of Test SATD. We employ GELU for the activation function and AdamW for the optimizer. The maximum token length is set to 128, the batch size to 16, and the maximum number of epochs to 20. To prevent overfitting [69] during training, we employ early stopping [76], which halts training when the validation loss does not improve for five consecutive epochs.

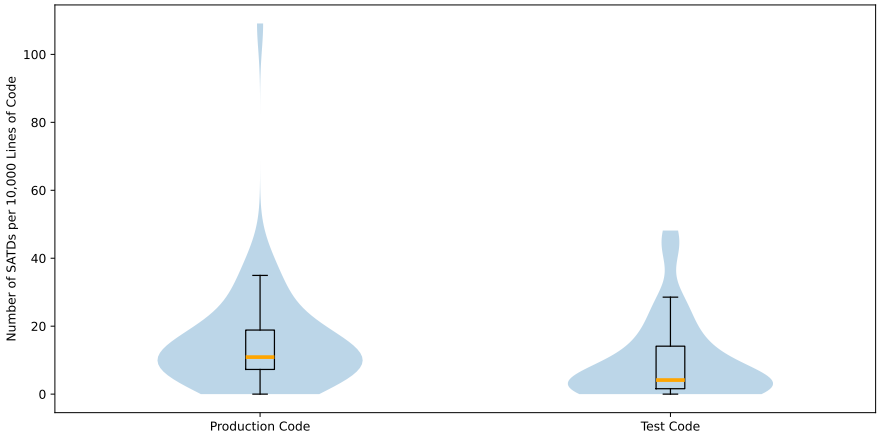



Fig. 3. Violin Plot of # SATD/10kLOC in Production and Test Code

Furthermore, we measure the performance of Large Language Models (LLMs) for Test SATD classification. This was inspired by a recent study by Li *et al.* [41], which investigates the capabilities of generative models such as ChatGPT for SATD detection. We constructed classifiers using two models via the OpenAI API: `GPT-3.5-turbo` and `GPT-4.1`. Unlike the fine-tuning approach used for CodeBERT, we employed a few-shot prompting strategy. For each SATD instance to be classified, the prompt included a description of our five SATD categories along with typical examples of each to guide the model's reasoning. The performance of these LLM-based classifiers was evaluated on the same dataset using the same metrics to ensure a direct comparison with our other models.

To assess model performance, we employed 10-fold cross-validation [22], a widely adopted technique for robust evaluation. The dataset was randomly partitioned into ten equally sized folds. In each iteration, nine folds were used for training and the remaining fold for testing, ensuring that each fold served as the test set exactly once. For LLM-based classifiers (GPT-3.5-turbo and GPT-4.1), which do not require training data, we applied the same 10-fold partitioning scheme but used only the test folds for evaluation. Across all models, we measure performance using three classic metrics: Precision, Recall, and F1-score. The F1 score is the harmonic mean of Precision and Recall. Since there is a trade-off between Precision and Recall, the F1 score evaluates the balance between Precision and Recall. In other words, it assesses whether the increase in Precision (or Recall) outweighs the decrease in Recall (or Precision).

## 5 Results

### 5.1 $RQ_1$: How prevalent is SATD in test code?

By applying the SATD detection tool `DebtHunter`, we obtained a total of 14,987 and 2,779 SATD instances in the production and test code, respectively. SATD in test code accounts for 15.6% of all detected SATD, which differs from the result reported in Bavota *et al.* [7]. Their finding indicates that test-related SATD accounts for only 8%, significantly lower than our results. This discrepancy may be caused by three factors. First, we collected SATD instances from different repositories, and half of them are used in testing-related studies, which might indicate that the repositories are well-tested. Second, as the repositories in the previous study were collected in 2015 (those in ours collected in 2024), writing tests have become more common recently [31, 33]. Third, the previous studies manually analyzed whether the SATD is directly related to testing activities, while we count

SATD instances in test code instead. For example, if SATD is related to software implementation, it will not fall into the test category even if it is located in the test code.

As the lines of code in the production code and test code differ significantly, we also compare the normalized number of SATD instances. Figure 3 depicts the distribution of the numbers of SATD per 10,000 lines of code for each repository. In these violin plots, the thickness of the outer layer represents the probability density of the plotted values. In the center of each violin plot, the yellow line shows the median while the box bar represents the interquartile range. When looking at the quartiles, the number of SATD instances per 10k LOC range from 7.3 to 18.9 in the production code, while from 1.6 to 14.1 in the test code. The average and median values for Production SATD are also much higher than that for Test SATD (average: 15.6 vs. 9.5, median: 10.9 vs. 4.1). In terms of median, the normalized number of Test SATD is 62.4% smaller than that of the Production SATDs. A statistically significant difference is also observed with the Wilcoxon signed-rank test (p-value: 0.00006). In addition, the effect size (Cliff's Delta) was found to be 0.365, indicating a medium difference. Table 2 presents the detailed results of the statistical test.

**RQ1. While there are fewer Test SATD instances than Production SATD (*i.e.,* 10.9 and 4.1, respectively), the number is still non-negligible in the studied repositories.**

Table 2. Summary of Key Statistical Test Results

| | Metrics | Statistical Test | p-value | Effect Size |
|---|---|---|---|---|
| **RQ1** | SATD Density | Wilcoxon signed-rank | $< 0.001$ | Medium (d = 0.365) |
| **RQ2** | LOC | Mann-Whitney U | $< 0.001$ | Negligible (r = 0.009) |
| | Assertions | Mann-Whitney U | 0.48 | Negligible (r = 0.009) |
| | Annotations | Mann-Whitney U | $< 0.001$ | Negligible (r = 0.006) |
| | Cyclomatic Comp. | Mann-Whitney U | $< 0.001$ | Small (r = 0.014) |
| | Readability | Mann-Whitney U | $< 0.001$ | Negligible (r = 0.006) |
| | Test Smells | Mann-Whitney U | 0.08 | Negligible (r = 0.002) |
| | Code Smells | Mann-Whitney U | $< 0.001$ | Small (r = 0.013) |

### 5.2 $RQ_2$: Is SATD in test code correlated with test quality?

We calculated test quality metrics for test methods with and without Test SATD. Figure 4 depicts the distributions of each metric based on the presence or absence of Test SATDs. Table 2 presents the detailed results of this test, including the p-value and effect size.

For lines of code (LOC) (Figure 4a), test methods with SATD had a median LOC of 8, which is slightly higher than those without SATD (median LOC: 7). We applied the Mann-Whitney U test to these two groups and observed a statistically significant difference ($p = 2.32 \times 10^{-10}, r = 0.0094$). Note that the calculation of LOC excludes comments, indicating that test methods with SATD are simply larger.

Regarding the number of assertions (Figure 4b), the medians were both 0, and no statistically significant difference was observed ($p = 0.48, r = 0.00092$). However, for the number of annotations (Figure 4c), the medians were both 1, but on average, test methods with SATD had 0.79 annotations, whereas those without SATD had 0.74 annotations. The Mann-Whitney U test identified a statistically significant difference ($p = 2.40 \times 10^{-6}, r = 0.0059$).

Regarding readability (Figure 4d), we found a counter-intuitive result: the median for test methods with SATD was 0.60, which is slightly higher than those without SATD (*i.e.,* 0.59). A statistically significant difference was observed with the Mann-Whitney U test ($p = 4.02 \times 10^{-5}$, $r = 0.006$).

For cyclomatic complexity (Figure 4e), while the median was 1 for both groups, the mean for test methods with SATD was 1.72, compared to 1.29 for methods without SATD. The Mann-Whitney U test confirmed a statistically significant difference ($p = 4.95 \times 10^{-48}, r = 0.014$), clarifying that methods with Test SATD tend to be structurally more complex.

For the number of test smells (Figure 4f), the medians were both 0, and no statistically significant difference was observed by the Mann-Whitney U test ($p = 0.08, r = 0.0022$). Additionally, we conducted Chi-square test to examine the relationship between the presence of Test SATD and test smells in methods. The observed frequencies of Test SATD and test smells are summarized in Table 3. The chi-square test yielded $\chi^2(1, N = 447,040) = 1.19, p = 0.28$, indicating no statistically significant association between the presence of Test SATD and the presence of test smells.

Table 3. Confusion matrix of test methods with/without Test SATDs and **Test** Smells

| | With Test Smell | Without Test Smell |
|---|---|---|
| With SATD | 854 | 1,761 |
| Without SATD | 149,721 | 294,704 |

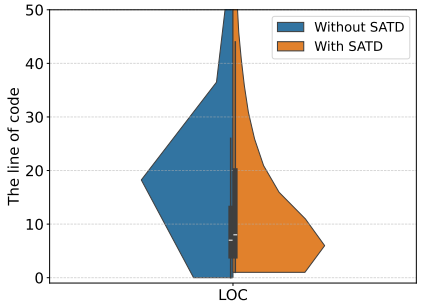
On the other hand, for the number of code smells, although the medians were both 0, the Mann-Whitney U test showed a p-value of $1.36 \times 10^{-24}$, indicating a statistically significant difference with a negligible effect ($r = 0.013$). This confirms that methods with Test SATD tend to have a statistically higher number of code smells, but the difference is not large. We also performed the Chi-square test to investigate the relationship between the presence of Test SATD and code smells (the metrics are summarized in Table 4). The chi-square test yielded $\chi^2(1, N = 447,040) = 49.3, p = 2.25 \times 10^{-12}$, indicating that there is a statistically significant difference. This contrast between the findings for test smells and code smells is one of the key insights of our study. Our results show that while there is a statistically significant correlation between the presence of Test SATD and the number of code smells, we found no such correlation with the number of test smells. This implies that Test SATDs are more likely to happen in test methods that have code quality issues rather than test-specific design problems. In other words, Test SATD and test smells represent different facets of quality issues in test code and can occur independently.

Table 4. Confusion matrix of test methods with/without Test SATDs and **Code** Smells

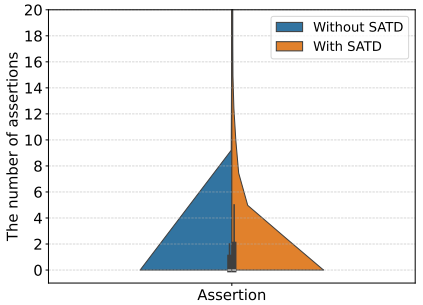
| | With Code Smell | Without Code Smell |
|---|---|---|
| With Test SATD | 1,041 | 1,574 |
| Without Test SATD | 147,996 | 296,429 |

Furthermore, we conducted a correlation analysis among these quality metrics. We calculated Spearman's rank correlation coefficient for all pairs of the seven metrics, separately for test methods with and without SATD. The Spearman coefficient ($\rho$) ranges from -1 to +1, indicating the direction and strength of a monotonic relationship. A positive value indicates that as one metric increases, the other tends to increase, while a negative value indicates that as one metric increases, the other tends to decrease. Figure 5a and 5b visualize these correlation coefficients as two heatmaps.

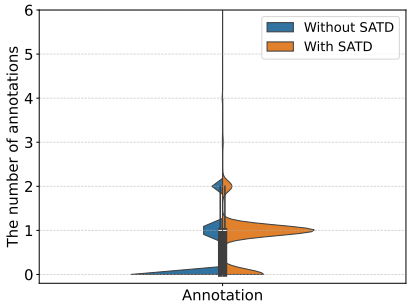
Overall, these two heatmaps present similar correlation trends between the two groups. However, we observed notable differences in the correlations involving the number of annotations. The largest difference was the correlation between Annotations and LOC, which was weakly positive in methods without SATD ($\rho = 0.288$) but near zero in methods with SATD ($\rho = -0.013$), leading to a difference of 0.301. Similarly, the correlation between Annotations and Assertions was also considerably weaker in methods with SATD ($\rho = 0.059$) compared to those without ($\rho = 0.267$),



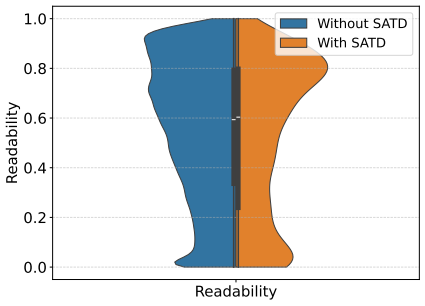

(a) LOC



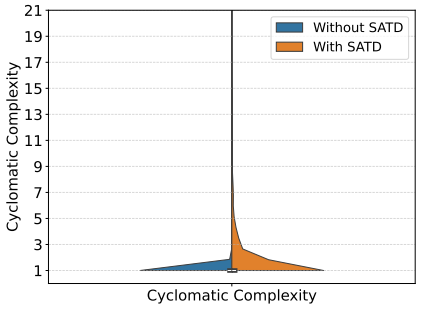

(b) Assertions



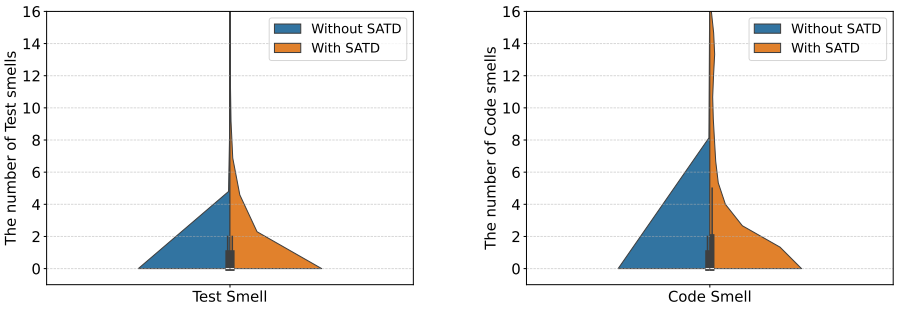

(c) Annotations




(d) Readability



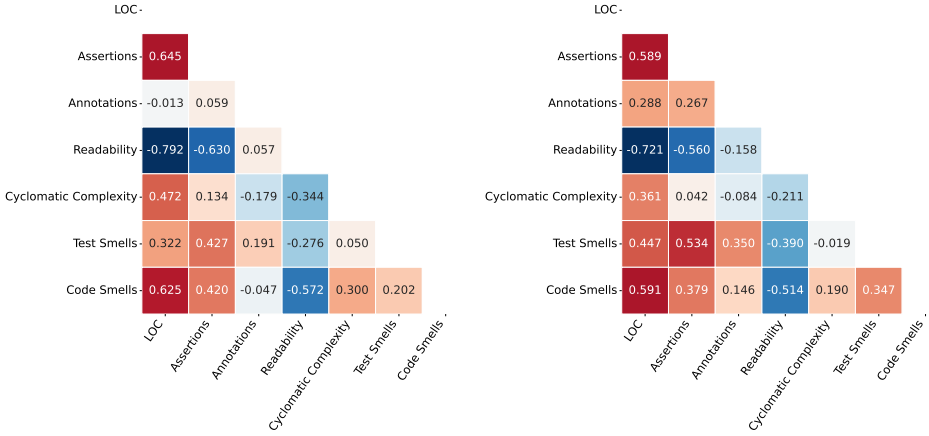

(e) Cyclomatic Complexity




(f) Test smells (g) Code smells

Fig. 4. Distribution in software quality metrics measured in methods with and without Test SATD

with a difference of 0.208. These findings suggest that the role of annotations may change when developers introduce SATD; in normal test methods, more annotations may be associated with longer code and more assertions, but this relationship appears to break down in the presence of technical debt.




(a) With SATD (b) Without SATD

Fig. 5. Spearman Correlation Matrices with and without SATD

**RQ2. Test methods with SATD have more lines of code, annotations, code smells, readability, and complexity than those without SATD. However, there is no statistical correlation between Test SATD and test smells, implying that these issues occur independently.**

Table 5. Classification of Test SATD

| Main Cat. | Sub Cat. | Detailed Category | # |
|---|---|---|---|
| Production-originated issues | Failures | Indicates the failure-triggering environments/settings | 11 |
| | | Indicates the failure-triggering inputs | 4 |
| | On-hold Tasks | Indicates the existence of defects in production code | 42 |
| | | Indicates the unimplemented production code | 18 |
| | | Specifies tasks to execute for debugging | 7 |
| Test-originated issues | Test Completeness | Indicates incomplete or unimplemented tests | 120 |
| | | Asks for extra test cases | 17 |
| | Test Design and Implementation | Workaround for impossible/difficult implementations | 53 |
| | | Indicates non-optimal way of test implementation | 44 |
| | | Provides an alternative way to implement/design the test | 37 |
| | | Doubts on specific test case design | 14 |
| | | Doubts on test implementation | 9 |
| | | Indicates the non-functional issues of tests | 6 |
| | Test Maintenance | Asks for test updates in response to future/latest software versions | 17 |
| | | Asks for test code restructuring/refactoring | 9 |
| | | Asks for add/update documentation | 7 |
| | | Indicates flaky tests | 5 |
| | | Asks for updates of used production code in tests | 4 |
| | | Asks for test deletion | 4 |
| | | Indicates invalid or unused tests | 2 |

### 5.3 $RQ_3$: What are the purposes of SATD in test code?

We inspected 506 randomly selected SATD comments. The process of building the taxonomy, as detailed in Section 4.2.3, was an iterative effort of labeling and merging. The initial fine-grained labeling of the first batch of 100 instances yielded 146 distinct labels, which were then consolidated into 50 initial categories after a reconciliation meeting. After the full set of 506 instances was classified through the iterative process, a total of 64 categories had been identified. A final round of discussion was then conducted to merge and refine these, resulting in the final taxonomy of 20 categories. Of the 506 instances, 407 instances (80.4%) are assigned with consistent labels from both reviewers, resulting in a Cohen's kappa coefficient of 0.78, indicating a substantial agreement according to Landis and Koch [40]. The conflicts were resolved with a third annotator. Throughout the manual inspection, we found that 36 instances were false positives (incorrectly detected as SATD) and the label of 40 instances could not be assigned due to the lack of information. As a result, 430 SATD instances were included in the final results. Table 5 shows the final classification of Test SATD and the breakdown of each category. In Table 6, we identified the top five most frequent words for each category except typical SATD tags such as `TODO` and `FIXME`. In particular, we split the words in SATD comments, removed stopwords using the NLTK library,[10] and identified the most frequent words for each category.

In the following, we introduce each category, separated by the origin of issues (whether the issue originated from production code or test code).

[10] https://www.nltk.org

Table 6. Most frequent words by subcategory

| Subcategory | Common words |
|---|---|
| Failures | `fails, test, enabled, passes, macos` |
| On-hold Tasks | `currently, supported, yet, work, fix` |
| Test Completeness | `assert, value, test, check, exception` |
| Test Design and Implementation | `test, workaround, need, way, better` |
| Test Maintenance | `remove, version, buffer, response, future` |

*5.3.1 Production-originated Issues.* This category includes issues intertwined with production code or issues caused by the production code. A total of 82 Test SATD instances fall into this category. Production-originated issues are further divided into the following subcategories:

**Failures:** This subcategory of SATD indicates the causes of test failures, which are related to either environments/settings or inputs in the production code. 15 cases fell into this category, with 11 related to environments/settings and 4 related to specific inputs. The two most frequently words are `fails` and `enabled`, which are related to test execution status. Bavota *et al.* [7] also introduced examples of such failures, but their "Test debt" category does not contain any subcategories. Similarly, the study by Kashiwa *et al.* [36] also presents a "Failure" category for test-related SATD, but no further categorization was given. We separate the "failure"-related SATD into two types:

- **Indicates the failure-triggering environments/settings:** This type of SATD is used to notify developers that a test will fail with a specific environment or configuration. In many cases, developers comment out the assertions that fail or disable the whole test method. Snippet 1 shows an example, including an access-related failure occurring only on Linux and Windows.[11] This test method is enabled to run only on Mac OS, using "`@EnabledOnOs`" annotation.[12]

Listing 1. Example of SATD categorized into "Indicates the failure-triggering environments/settings"

```
1     /**
2      * TODO Passes on macOS, fails on Linux and Windows with
       ↪  AccessDeniedException.
3      */
4     @Test
5     @EnabledOnOs(value = OS.MAC)
6     public void testForceDeleteUnwritableDirectory() throws Exception {
```

- **Indicates the failure-triggering inputs:** This type of SATD notifies developers that the test will fail when the production code receives a specific input, which is illustrated in Snippet 2. In the example, the test is commented out due to a failure caused by a specific file.[13]

Listing 2. Example of SATD categorized into "Indicates the failure-triggering inputs"

```
1     public void testReadCompressedAndroMDAProfileIssue5946() {
2         // TODO: uncomment the following to get the failure.
3 //        assertLoadModel(ISSUE5946_BASE_DIR + "zipped-uml14"
```

[11] https://github.com/apache/commons-io/blob/290d72eda9152d1e11b79d48453908ff3f6b9897/src/test/java/org/apache/commons/io/FileUtilsTest.java#L1730

[12] https://junit.org/junit5/docs/5.2.0/api/org/junit/jupiter/api/condition/EnabledOnOs.html

[13] https://github.com/argouml-tigris-org/argouml/blob/6b6db0242a40f80655cbfdddcca246afe23df20c/src/argouml-core-model-mdr/tests/org/argouml/model/mdr/TestReadCompressedFilesAndHref.java#L73

```
//            +
↪ "/andromda-profile-datatype/3.3/andromda-profile-datatype-3.3.xml.zip",
//            null, "testReadCompressedAndroMDAProfileFileIssue5946");
    }
```

**On-hold Tasks:** This subcategory of SATD indicates a test is on hold until a specific task in production code is executed, such as new implementation or bug fixing. The prevalence of words such as `currently`, `yet` shows that developers view these issues as temporary states with an expectation of future resolution. The "On-hold SATD" is originally defined by Maipradit *et al.* [44], which analyzed on-hold SATD in the production code without taking into account the tests. We identified 67 cases in this category and the on-hold tasks in test code include the following types:

- **Indicates the existence of defects in production code:** This type of SATD is used to inform developers of defects present in the production code. We identified 42 instances of this type, making it the most prevalent on-hold task. Snippet 3 shows an example of this category, where the test is currently disabled until the defect in the YARN project is fixed.[14]

Listing 3. Example of SATD categorized into "Indicates the existence of defects in production code"

```
    // FIXME:
    // Disabled this test because currently, when shutdown hook triggered
    ↪ at
    // lastRetry in RM view, cleanup will not do. This should be supported
    ↪ after
    // YARN-2261 completed
  //   @Test (timeout = 30000)
     public void testDeletionofStagingOnKillLastTry() throws IOException {
```

- **Indicates the unimplemented production code:** This type of SATD indicates that a developer is waiting for the implementation of certain production code. An example can be seen in Snippet 4, in which a method call is commented out because a function for a property is not yet implemented.[15]

Listing 4. Example of SATD categorized into "Indicates the unimplemented production code"

```
        si.setTemplate(P_TEMPLATE);
        // FIXME (byte array properties not yet implemented):
        ↪ si.setThumbnail(P_THUMBNAIL);
        si.setTitle(P_TITLE);
```

- **Specifies tasks to execute for debugging:** This type of SATD informs developers of the specific task to execute for debugging. As illustrated in Snippet 5, the SATD comment requests an investigation into why the internal state is not visible during retry processing.[16]

[14] https://github.com/apache/hadoop/blob/bd8b77f398f626bb7791783192ee7a5dfaeec760/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-app/src/test/java/org/apache/hadoop/mapreduce/v2/app/TestStagingCleanup.java#L224-L227

[15] https://github.com/apache/poi/blob/ae2f0945cd2ab37260e46ab46c54b8f68a131aea/poi/src/test/java/org/apache/poi/hpsf/basic/TestWriteWellKnown.java#L230

[16] https://github.com/apache/camel/blob/66736471db8ddc22e50cc31c87d34b072455b488/components/camel-aws/camel-aws-xray/src/test/java/org/apache/camel/component/aws/xray/ErrorTest.java#L43

Listing 5. Example of SATD categorized into "Specifies tasks to execute for debugging"

```
1    // FIXME: check why processors invoked in onRedelivery do not generate a
     ↪  subsegment
2    public ErrorTest() {
```

*5.3.2 Test-originated Issues.* This category includes issues originating from the test code. 348 relevant Test SATD instances are identified, which can be further divided into the following subcategories:

**Test Completeness:** This subcategory of SATD pertains to the effectiveness and completeness of test code, such as incomplete test implementations, insufficient test cases, or unclear intent of test cases. The category featured words related to test case verification, such as `check` and `assert`. A previous study [36] introduced the "Necessity" subcategory, which corresponds to our "Indicates incomplete or unimplemented tests" or "Asks for extra test cases" categories. 137 cases were identified in this category, with the most frequent one (120 cases) being "Indicates incomplete or unimplemented tests". The subcategories in this category are described as follows.

- **Indicates incomplete or unimplemented tests:** This type of SATD points out that the test is either incomplete or unimplemented, leading to invalid or ineffective test cases. Snippet 6 shows an example of this subcategory, in which an assertion is needed to check the value produced by the previous line.[17]

Listing 6. Example of SATD categorized into "Indicates incomplete or unimplemented tests"

```
1    @Test(expected = BuildException.class)
2    public void testElementCreatorTwo() {
3        ih.getElementType("two");
4        // TODO we should be asserting a value in here
5    }
```

- **Asks for extra test cases:** This type of SATD indicates the need for extra test cases. In contrast to "Indicates incomplete or unimplemented tests", this type of SATD requests implementing entire test methods rather than completing existing test methods. They are often created to further ensure the behavior is correct or to improve test coverage. Snippet 7 shows an example where an additional test case is required to handle invalid two-byte pairs.[18]

Listing 7. Example of SATD categorized into "Asks for extra test cases"

```
1    @Test
2    public void validateUDTNested()
3    {
4        validate(nestedUDTGen());
5    }
6
7    // todo: for completeness, should test invalid two byte pairs.
```

[17] https://github.com/apache/ant/blob/53f19eccf49acf526415997046dca5a5135b0e8f/src/tests/junit/org/apache/tools/ant/IntrospectionHelperTest.java#L128

[18] https://github.com/apache/cassandra/blob/6b134265620d6b39f9771d92edd29abdfd27de6a/test/unit/org/apache/cassandra/db/marshal/TypeValidationTest.java#L256

**Test Design and Implementation:** This subcategory of SATD is related to the design and implementation of test code. 163 cases fall into this category. The appearance of words like `workaround` and `better` indicates that this category captures SATD related to suboptimal implementations or design improvements needed in tests. The closest categories from related work are "Code debt" and "Design Debt" from Bavota *et al.*'s [7]. In their subcategories, they also mentioned similar SATD types related to "workaround", "code quality", and "design". However, their categories focus on the production code, while we investigate test code. In the test code, the relevant SATD can be further categorized as follows:

- **Indicates workaround for impossible/difficult implementations:** This type of SATD is used to inform developers about the workaround for code either impossible or difficult to implement. This is the most frequent SATD type (53 instances) regarding Test Design and Implementation. An example can be seen in Snippet 8, in which a developer attempted to retrieve the value of a property, but the current implementation of the production code prevents this due to encapsulation. Consequently, the developer resorted to using reflection to obtain the value. The comment clarifies the rationale behind this implementation choice.[19]

Listing 8. Example of SATD categorized into "Indicates workaround for impossible/difficult implementations"

```
1    @Test
2    public void testProperties() throws Exception {
3        // reflection hack ... no other way to get raw props ...
4        Field configField =
         ↪  KafkaStreams.class.getDeclaredField("applicationConfigs");
5        configField.setAccessible(true);
6        StreamsConfig config = (StreamsConfig) configField.get(streams);
7        ...
```

- **Indicates non-optimal way of test implementation:** This type of SATD is used to highlight non-optimal implementations. Snippet 9 provides an example of such a case, where a comment points out the use of a method or approach that is suboptimal. In this instance, the SATD comment indicates that directly using `DataFormatReifier` is not the best practice and suggests that a better approach should be considered.[20]

Listing 9. Example of SATD categorized into "Indicates non-optimal way of test implementation"

```
1    private FhirDataFormat getDataformat(String name) {
2        CamelContext camelContext = context();
3        // TODO: Do not use reifier directly
4        return (FhirDataFormat)
         ↪  DataFormatReifier.getDataFormat(camelContext, name);
5    }
```

[19] https://github.com/quarkusio/quarkus/blob/a91a36c533676a5d35ddedbef1717392c9191360/integration-tests/kafka-streams/src/test/java/io/quarkus/it/kafka/streams/KafkaStreamsPropertiesTest.java#L26

[20] https://github.com/apache/camel/blob/66736471db8ddc22e50cc31c87d34b072455b488/components/camel-fhir/camel-fhir-component/src/test/java/org/apache/camel/component/fhir/dataformat/spring/FhirDataformatConfigSpringTest.java#L90

- **Provides an alternative way to implement/design the test:** This type of SATD is used to propose a different way to implement the test. Snippet 10 provides an example of this type of SATD, where the comment suggests using "interrupts" instead of `notifyAll()` to target waiting threads more effectively.[21]

Listing 10. Example of SATD categorized into "Provides an alternative way to implement/design the test"

```
    if (waitingOnRelinquish)
    {
        waitingOnRelinquish = false;
        monitor.notifyAll(); // TODO: could use interrupts to target waiting
        ↪ anyway, avoiding notifyAll()
    }
```

- **Doubts on specific test case design:** This type of SATD raises questions about purposes, validity or effectiveness of test cases. Snippet 11 provides an example where a developer questioned the purpose of the test case and subsequently commented out the entire test method.[22]

Listing 11. Example of SATD categorized into "Doubts on specific test case design"

```
    // TODO: what is this test?
//    @Test
//    public void testRecoverLimboFlushFailure() throws Exception {
//        MockLedgerManager lm = new MockLedgerManager();
          ...
```

- **Doubts on test implementation:** This type of SATD appears when developers are unclear about whether the implementations are optimal or not. Snippet 12 provides an example, in which the comment raises questions about the necessity of the `Thread.sleep()` call, suggesting that a further review is needed.[23]

Listing 12. Example of SATD categorized into "Doubts on test implementation"

```
    mockResultEndpoint.expectedBodiesReceivedInAnyOrder(commitComment1,
    ↪ commitComment2);

    Thread.sleep(1 * 1000);          // TODO do I need this?

    mockResultEndpoint.assertIsSatisfied();
```

- **Indicates the non-functional issues of tests:** This type of SATD is used to inform developers about non-functional issues in the test code, such as performance, scalability, or resource usage. Snippet 13 provides an example, where the comment indicates that the test takes an excessive amount of time to execute, potentially impacting overall test efficiency.[24]

[21] https://github.com/apache/cassandra/blob/6b134265620d6b39f9771d92edd29abdfd27de6a/test/simulator/main/org/apache/cassandra/simulator/systems/InterceptingMonitors.java#L363
[22] https://github.com/apache/bookkeeper/blob/cce4b6461691466c663f2cb4d00dd4d73dd9071e/bookkeeper-server/src/test/java/org/apache/bookkeeper/bookie/datainteg/DataIntegrityCheckTest.java#L410
[23] https://github.com/apache/camel/blob/66736471db8ddc22e50cc31c87d34b072455b488/components/camel-github/src/test/java/org/apache/camel/component/github/consumer/PullRequestCommentConsumerTest.java#L55
[24] https://github.com/cloudfoundry/uaa/blob/5b74878c0297043860bb88434ee1123934acfe19/server/src/test/java/org/cloudfoundry/identity/uaa/util/CachingPasswordEncoderTest.java#L125

Listing 13. Example of SATD categorized into "Indicates the non-functional issues of tests"

```
1    @Test
2    // TODO: This test takes a long time to run :(
3    void ensureNoMemoryLeak() {
```

**Test Maintenance:** This subcategory of SATD is related to test code maintenance. These SATD instances indicate the need for updates in future versions, refactoring, fixing flaky tests, and adding documentation. Frequent terms such as `remove` and `version` indicate that this category captures SATD related to test code management, including cleanup and version updates. This subcategory does not exist in Bavota *et al.*'s work [7], but "Asks for test updates in response to future/latest software versions" partially matches the "Future work" subcategory in Kashiwa *et al.*'s study [36]. We identified 48 cases in this category, with the most frequent subcategory being "Asks for test updates in response to future/latest software versions," represented by 17 instances. The details of each subcategory are described below.

- **Asks for test updates in response to future/latest software versions:** This type of SATD is used to inform developers that the test needs to be updated in a future version of production code. Snippet 14 provides an example of this subcategory. The comment suggests the removal of the reference to SecurityManager in this test case when the project updates the version of the programming language Groovy, as the class is discontinued in the new version.[25]

Listing 14. Example of SATD categorized into "Asks for test updates in response to future/latest software versions"

```
1    @SuppressWarnings("removal") // TODO in a future Groovy version remove
     ↪ reference to SecurityManager, for now not run for JDK18+
2    public void testInvokesPrivateMethodsInGroovyObjectsWithoutChecks()
     ↪ throws Exception {
3        if (isAtLeastJdk("18.0")) return;
4        ...
```

- **Asks for test code restructuring/refactoring**: This type of SATD is used to notify developers that the test code needs to be restructured or refactored. Snippet 15 shows an example, where the SATD comment suggests making the contract more explicit or extracting common code to improve readability and maintainability.[26] When looking into 9 cases, most of the SATDs request structural refactoring, including five instances of Move Method/Class, two of Extract Method, one of Rename Method, and one involving data structure changes (*e.g.,* Introduce Parameter Object). We also found that only one case co-occurred with a test smell (specifically, "Assertion Roulette"), reinforcing our finding that Test SATD and test smells are largely independent issues.

[25] https://github.com/apache/groovy/blob/f6221ee780bfb2f84fb197da2c13387c4e93a019/src/test/org/codehaus/groovy/reflection/SecurityTest.java#L301
[26] https://github.com/apache/flink/blob/eaffd227d853e0cdef03f1af5016e00f950930a9/flink-state-backends/flink-statebackend-changelog/src/test/java/org/apache/flink/state/changelog/ChangelogStateDiscardTest.java#L376

Listing 15. Example of SATD categorized into "Asks for test code restructuring / refactoring"

```
1           // todo: make the contract more explicit or extract common code
2           Map<UploadTask, Map<StateChangeSet, Tuple2<Long, Long>>>
            ↪  taskOffsets =
3                   tasks.stream().collect(toMap(identity(),
                    ↪  this::mapOffsets));
```

- **Asks for add/update documentation:** This type of SATD is used to request additional documentation or updates to the documentation to understand the goals and implementation of the test code. Snippet 16 provides an example, where a comment highlights the need for documentation to clarify specific behavior.[27]

Listing 16. Example of SATD categorized into "Asks for add/update documentation"

```
1   // TODO document this behaviour.
2   // Is it different AspectJ behaviour, at least for checked exceptions?
3   @Test
4   void aspectMethodThrowsExceptionIllegalOnSignature() {
5       TestBean target = new TestBean();
6       RemoteException expectedException = new RemoteException();
7       List<Advisor> advisors = getAdvisorFactory().getAdvisors(
8           aspectInstanceFactory(new
            ↪  ExceptionThrowingAspect(expectedException), "someBean"));
9       assertThat(advisors).as("One advice method was found").hasSize(1);
10      ITestBean itb = createProxy(target, ITestBean.class, advisors);
11      assertThatExceptionOfType(UndeclaredThrowableException.class)
12          .isThrownBy(itb::getAge)
13          .withCause(expectedException);
14  }
```

- **Indicates flaky tests:** This type of SATD informs developers that the test is unstable—sometimes passing and sometimes failing (*i.e.,* flaky tests) [23]. Snippet 17 provides an example of such SATD. In this instance, the comment warns that the implementation causes flaky tests.[28]

Listing 17. Example of SATD categorized into "Indicates flaky tests"

```
1   /**
2    * TODO Fails randomly.
3    */
4   @Test
5   public void testWaitForInterrupted() throws InterruptedException {
```

- **Asks for updates of used production code in tests:** This type of SATD is used to indicate outdated production code used in the test. Snippet 18 provides an example, where a comment requests updating the test to use a new conflict resolver.[29]

[27] https://github.com/spring-projects/spring-framework/blob/f85d5bd84a7e7bc810bb5a8179fc2fc130affc89/spring-aop/src/test/java/org/springframework/aop/aspectj/annotation/AbstractAspectJAdvisorFactoryTests.java#L435
[28] https://github.com/apache/commons-io/blob/290d72eda9152d1e11b79d48453908ff3f6b9897/src/test/java/org/apache/commons/io/FileUtilsWaitForTest.java#L42
[29] https://github.com/apache/maven/blob/8b094c9513efc1b9ce2d952b3b9c8eaedaf8cbf0/maven-compat/src/test/java/org/apache/maven/repository/legacy/resolver/DefaultArtifactCollectorTest.java#L155

Listing 18. Example of SATD categorized into "Asks for updates of used production code in tests"

```
public void
↪ disabledtestResolveCorrectDependenciesWhenDifferentDependenciesOnNewest()
        throws ArtifactResolutionException,
        ↪ InvalidVersionSpecificationException {
    // TODO use newest conflict resolver
    ArtifactSpec a = createArtifactSpec("a", "1.0");
    ArtifactSpec b = a.addDependency("b", "1.0");
    ArtifactSpec c2 = b.addDependency("c", "2.0");
    ArtifactSpec d = c2.addDependency("d", "1.0");
```

- **Asks for test deletion:** This type of SATD is used to report unnecessary tests. Snippet 19 provides an example, where a comment requests the deletion of a test that is no longer needed.[30]

Listing 19. Example of SATD categorized into "Asks for test deletion"

```
// TODO remove this test
@Test
void testStaticConstructor() throws NoSuchFieldException {
```

- **Indicates invalid or unused tests:** This type of SATD is used to inform other developers that the test is currently invalid or unused. Snippet 20 provides an example of such SATD, where the comment asks other developers to verify if the test is unused.[31]

Listing 20. Example of SATD categorized into "Indicates invalid or unused tests"

```
/**
 * @throws SAXException when things go wrong with SAX
 * @throws IOException when there's an IO error
 * @throws ParserConfigurationException when the parser finds wrong
 ↪ syntax
 *
 * TODO: Unused?
 */
public void testDataModel()
    throws SAXException,
           IOException,
           ParserConfigurationException {
```

**RQ3. Test SATD serves various purposes and can be classified into five major categories related to 20 types of issues. The most frequent purpose of Test SATD is "Indicating incomplete or unimplemented tests". Additionally, most of the issues identified in this study do not fit the existing categories proposed by previous studies, highlighting the differences between SATD in the production and test code.**

[30] https://github.com/apache/dubbo/blob/3609ddb2259ad223f6c0a827e36f6f8ccd38c6b2/dubbo-config/dubbo-config-api/src/test/java/org/apache/dubbo/config/MethodConfigTest.java#L111
[31] https://github.com/argouml-tigris-org/argouml/blob/6b6db0242a40f80655cbfdddcca246afe23df20c/src/argouml-app/tests/org/argouml/model/TestAgainstUmlModel.java#L91

### 5.4 $RQ_4$: To what extent can SATD in Test Code be classified automatically?

Table 7 presents the results of classifying Test SATD into five subcategories using machine-learning and deep-learning classifiers. First, in terms of precision, the GPT-3.5-turbo model achieved the highest overall precision (*i.e.,* 0.77), indicating its effectiveness in preventing false positives. Conversely, the XGBoost classifier exhibited the lowest overall precision, with a value of 0.59.

When looking into the recall, deep-learning models overall outperformed traditional machine-learning models. The highest recall of 0.67 was jointly achieved by the fine-tuned CodeBERT model and the large language model GPT-4.1. The BERT-based classifier also performed well with a recall of 0.64. In contrast, the Naive Bayes classifier recorded the lowest recall at 0.39, resulting in a higher number of undetected instances.

In terms of F1-score, the CodeBERT-based classifier achieved the highest overall score (*i.e.,* 0.70), demonstrating robustness in accurately identifying relevant SATD instances while minimizing misclassifications. Notably, the superiority of CodeBERT over BERT suggests that characteristic words specific to test code are more prevalent than in natural language, a finding also observed in studies focusing on production code [62].

Figure 6 presents a Venn diagram illustrating the distribution of correctly classified instances among the top five classifiers by F1-score.[32] 185 out of 430 instances were correctly predicted by all the classifiers. Furthermore, this qualitative analysis highlights a crucial finding that the F1-score alone does not capture: the GPT-4.1 model made the most uniquely correct predictions (17 instances), surpassing all other models, including the top-performing CodeBERT. This strongly suggests that while CodeBERT is a robust, well-balanced classifier, modern LLMs like GPT-4.1 possess a distinct capability to understand different semantic nuances in SATD comments that other models may miss.

Next, looking into the prediction performance for subcategories, the best-performing model CodeBERT exhibits outstanding performance for predicting SATDs falling into "Test Completeness" category (*i.e.,* F1 score of 0.83). On the other hand, the lowest F1-score was observed in the "Failures" category with a value of 0.54. In particular, instances of the "Failures" category were frequently misclassified as "On-hold Task". For example, we observe several cases that have a trigger condition which is often used in "On-hold Task" category such as "`TODO: WriteResult isn't returned when inserting`".[33] This low performance is likely to be caused by the lack of instances (*i.e.,* the dataset contains only 15 instances from the "Failures" category). Our future work will extend the dataset to increase the number of SATD that fall into these smaller categories.

**RQ4. The CodeBERT-based model outperforms other machine learning models in terms of Recall and F1-Score (0.67 and 0.70, respectively). It achieves the highest performance in the category of "Test Completeness" (0.83), while showing the lowest performance in the "Failures" category (0.54). While the prediction results are reasonable, there is still large room for improvement.**

[32] Our Venn diagram does not include results from all eight classifiers as such plots are unreadable. Instead, plots with five classifiers are easier to comprehend.

[33] https://github.com/apache/camel/blob/66736471db8ddc22e50cc31c87d34b072455b488/components/camel-mongodb/src/test/java/org/apache/camel/component/mongodb/integration/MongoDbHeaderHandlingIT.java#L71

Table 7. Performance comparison using different machine-learning or deep-learning algorithms

(a) SVM (Accuracy: 0.63)

| | Failures | On-hold Task | Test Completeness | Test Design and Implementation | Test Maintenance | Average |
|---|---|---|---|---|---|---|
| Precision | 0.83 | 0.71 | 0.73 | 0.54 | 1.00 | 0.76 |
| Recall | 0.33 | 0.36 | 0.61 | 0.87 | 0.31 | 0.50 |
| F1-Score | 0.48 | 0.48 | 0.67 | 0.67 | 0.48 | 0.55 |

(b) Naive Bayes (Accuracy: 0.59)

| | Failures | On-hold Task | Test Completeness | Test Design and Implementation | Test Maintenance | Average |
|---|---|---|---|---|---|---|
| Precision | 0.00 | 0.87 | 0.57 | 0.56 | 1.00 | 0.60 |
| Recall | 0.00 | 0.19 | 0.73 | 0.78 | 0.25 | 0.39 |
| F1-Score | 0.00 | 0.32 | 0.64 | 0.65 | 0.40 | 0.40 |

(c) XGBoost (Accuracy: 0.61)

| | Failures | On-hold Task | Test Completeness | Test Design and Implementation | Test Maintenance | Average |
|---|---|---|---|---|---|---|
| Precision | 0.54 | 0.61 | 0.64 | 0.61 | 0.54 | 0.59 |
| Recall | 0.47 | 0.52 | 0.63 | 0.71 | 0.42 | 0.55 |
| F1-Score | 0.50 | 0.56 | 0.63 | 0.66 | 0.47 | 0.57 |

(d) Random Forest (Accuracy: 0.66)

| | Failures | On-hold Task | Test Completeness | Test Design and Implementation | Test Maintenance | Average |
|---|---|---|---|---|---|---|
| Precision | 0.67 | 0.76 | 0.73 | 0.58 | 0.91 | 0.73 |
| Recall | 0.27 | 0.37 | 0.69 | 0.85 | 0.42 | 0.52 |
| F1-Score | 0.38 | 0.50 | 0.71 | 0.69 | 0.57 | 0.57 |

(e) BERT (Accuracy: 0.74)

| | Failures | On-hold Task | Test Completeness | Test Design and Implementation | Test Maintenance | Average |
|---|---|---|---|---|---|---|
| Precision | 0.55 | 0.59 | 0.83 | 0.74 | 0.73 | 0.69 |
| Recall | 0.40 | 0.54 | 0.79 | 0.85 | 0.62 | 0.64 |
| F1-Score | 0.46 | 0.56 | 0.81 | 0.79 | 0.67 | 0.66 |

(f) CodeBERT (Accuracy: **0.77)**

| | Failures | On-hold Task | Test Completeness | Test Design and Implementation | Test Maintenance | Average |
|---|---|---|---|---|---|---|
| Precision | 0.64 | 0.65 | 0.85 | 0.77 | 0.72 | 0.73 |
| Recall | 0.47 | 0.63 | 0.82 | 0.86 | 0.60 | **0.67** |
| F1-Score | 0.54 | 0.64 | 0.83 | 0.81 | 0.66 | **0.70** |

(g) GPT-3.5-turbo (Accuracy: 0.59)

| | Failures | On-hold Task | Test Completeness | Test Design and Implementation | Test Maintenance | Average |
|---|---|---|---|---|---|---|
| Precision | 1.00 | 1.00 | 0.93 | 0.55 | 0.41 | **0.77** |
| Recall | 0.40 | 0.07 | 0.45 | 0.91 | 0.69 | 0.50 |
| F1-Score | 0.57 | 0.14 | 0.61 | 0.69 | 0.51 | 0.50 |

(h) GPT-4.1 (Accuracy: 0.73)

| | Failures | On-hold Task | Test Completeness | Test Design and Implementation | Test Maintenance | Average |
|---|---|---|---|---|---|---|
| Precision | 0.45 | 0.61 | 0.95 | 0.71 | 0.58 | 0.66 |
| Recall | 0.60 | 0.30 | 0.72 | 0.89 | 0.83 | **0.67** |
| F1-Score | 0.51 | 0.40 | 0.82 | 0.79 | 0.68 | 0.64 |

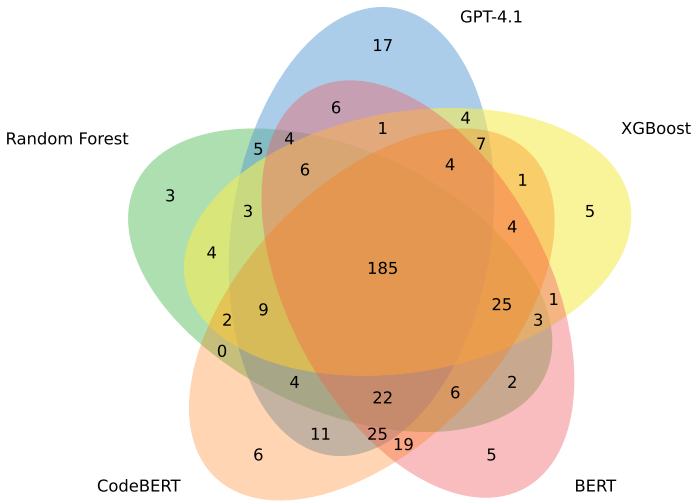



Fig. 6. Venn diagram of correct predictions for the top five classification models

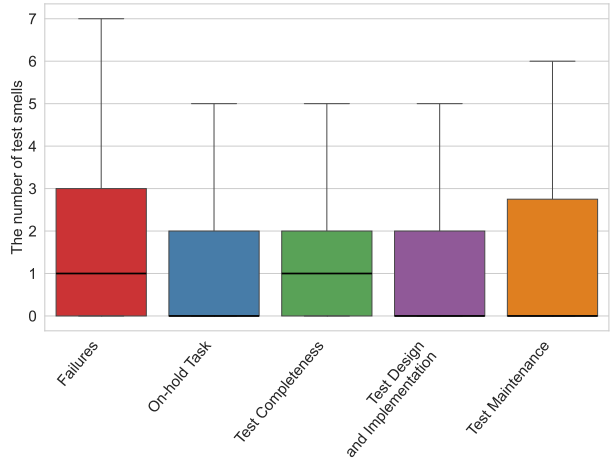



Fig. 7. The number of test smells by category

## 6 Discussions

### 6.1 Test Smells with Types of Test SATD

RQ2 examined the correlation between SATD in test code and various test quality indicators, including general code quality metrics (like code smells, lines of code, complexity, and readability) and test-specific metrics (such as test smells, assertions, and annotations) at the method level. RQ3 investigated the purposes and types of Test SATD, leading to a detailed taxonomy of 20 issue types categorized into five main groups.

While RQ2 found no direct statistical correlation between the general presence of Test SATD and test smells, it was not clear whether specific types of Test SATD identified in RQ3 might impact test quality, particularly regarding test smells. To examine this further, an additional analysis was conducted to study the impact of each type of SATD on quality aspects. This analysis leveraged the CodeBERT-based classifier, which was developed in RQ4 and demonstrated the highest overall performance (F1-score of 0.70) among the evaluated models for automatically classifying Test SATD types. This classifier was applied to all 2,092 filtered SATD instances to enhance the reliability and comprehensiveness of the analysis.

Figure 7 shows the distribution of the number of test smells across the SATD subcategories. Focusing on the median values, we observed that the "Test Completeness" and "Failures" categories had a median of 1, while the other categories had a median of 0. To determine whether there was a statistically significant difference in the number of test smells among the subcategories, we performed a Kruskal-Wallis test. As a result, we observed a statistically significant difference ($p < 0.05$). Furthermore, we conducted a post-hoc analysis using Dunn's Post-Hoc Test with a Bonferroni correction to deal with the family-wise error rate. We found that the "Failures" and "Test Completeness" categories (median=1) contained significantly more test smells than the other three categories ("On-hold Task," "Test Design and Implementation," and "Test Maintenance," all median=0), with $p < 0.05$.

### 6.2 Lessons Learned

This study investigates Test SATD to reveal its prevalence, relationship with quality metrics, and types. Additionally, we develop classifiers to categorize the types of Test SATD. Through this comprehensive study, we gained the following five key insights that inform both research and

Table 8. Comparison with the study of Bavota et al. [7]

| Aspect | Bavota et al. [7] | Our Study |
|---|---|---|
| *Studied Ecosystems* | 2 Ecosystems (Eclipse and Apache) | 24 Ecosystems (Apache, Spring, Gradle, etc.) |
| *Studied projects* | 159 projects | 50 projects |
| *Studied year* | 2016 | 2025 |
| *Scope of the study* | All Java code | All Java code (separated into prod. and test code) |
| *# Instances for qualitative analysis* | 366 Prod. SATD | 506 Test SATD |
| *% Test-related SATD* | 8% | 15.6% |
| *# Test Categories* | 1 category (*i.e.,* "Test" category) | 5 categories |
| *# Test Sub-categories* | None | 20 Sub-categories |

practice in test code quality management. Table 8 summarizes the differences in findings of the previous study and our study to facilitate our discussion.

**Lesson 1. Test SATD Is Distinct from Production SATD.**
A key lesson learned is that SATD in test code is not simply a secondary concern to production code SATD; it is a distinct phenomenon with unique characteristics and developer intentions. We identified 20 specific types of issues, categorized into five main groups, many of which do not directly align with existing categories for production code SATD. Notably, test-specific categories like "Doubts on specific test case design" emerged from our data but were absent even from general technical debt studies (*i.e.,* not only self-admitted technical debt). This divergence demonstrates that assuming similar characteristics for SATD across production and test code is an oversimplification.

Furthermore, we observed a significantly larger proportion of Test SATD (15.6%) compared to an 8% finding in a previous study from a decade ago. This temporal shift reflects the increasing emphasis on test writing in modern software development, underscoring the growing importance of addressing test code quality.

**Lesson 2. Developers Frequently Leave Incomplete Tests as SATD.**
The most frequent category of Test SATD identified was "Indicates incomplete or unimplemented tests" (120 out of 430 classified instances). This prevalence indicates a common developer practice of leaving SATD for missing assertions or entire test methods, even when the task might seem minor. A profound lesson here is the need to understand why developers choose to document these incomplete tasks as SATD rather than finishing them immediately. This behavior warrants further qualitative investigation through surveys or interviews to uncover the underlying reasons, such as time constraints, workflow interruptions, or perceived complexity of the remaining task. This insight also suggests a practical application: developers could benefit from specialized code completion tools tailored for test code, which might help address this specific form of technical debt proactively.

**Lesson 3. SATD Comments Reveal Issues Beyond Test Smells.**
Our study revealed a lack of a direct statistical correlation between Test SATD and test smells. This is a critical lesson: developers are explicitly flagging issues in test code that are often not detected by current automated test smell tools. This implies that existing test smell detection mechanisms are not fully capturing the spectrum of issues developers recognize and admit. This gap presents a clear opportunity for researchers and tool developers to leverage Test SATD comments as a rich, developer-centric source to identify new or overlooked types of test smells, thereby enhancing automated quality assurance tools.

**Lesson 4. Manual Classification Is Inherently Ambiguous.**
Our manual classification process, while rigorous, highlighted the inherent complexity in precisely categorizing developer intentions behind SATD comments. We employed a card sorting approach with independent labeling by two experienced annotators, and conflicts were resolved by a third, resulting in a Cohen's kappa coefficient of 0.78, indicating substantial agreement. This experience taught us that even with structured methods and expert annotators, ambiguity and misidentification can occur, suggesting that future efforts in SATD classification could benefit from more context-rich or interactive annotation processes.

**Lesson 5. Data Imbalance Hampers ML-Based SATD Classification.**
The performance of our machine learning models varied significantly across categories, with a noticeable drop for categories with fewer instances, such as the "Failures" category, which had only 15 instances in our dataset. While the CodeBERT-based classifier achieved the highest F1-score of 0.70 overall, its performance for "Failures" was the lowest at 0.54. This clearly demonstrates the impact of data imbalance on classification accuracy. A crucial lesson here is the need for a significantly larger and more balanced dataset to improve the robustness and accuracy of automatic SATD classification, especially for less frequent but potentially critical types of debt.

### 6.3 Implications

In this section, we discuss the implications for developers and researchers, mapping the findings to the research questions.

**Implication 1. Test SATD should not be considered as a negligible concern.**

RQ1 revealed that a non-negligible number of Test SATD instances exist in software repositories. Specifically, software testing plays a crucial role in modern software development, necessitating more proactive test writing than before [31, 33]. In fact, we observed a significantly larger ratio of Test SATD compared to a previous study conducted a decade ago [6], despite examining different projects. Many recent studies have reported that the quality of test code leads to serious quality issues in production [4, 65]. This suggests that *researchers should recognize the importance of studying SATD not only in production code but also in test code.* As previous studies have done for production code, future research should investigate the impact of Test SATD on reliability [74] and maintainability [53] of software.

**Implication 2. Test SATD is likely to be associated with code quality issues.**

RQ2 clarified that methods containing Test SATD have more lines of code (LOC), annotations, complexity, and code smells than methods without Test SATD. From the perspectives of LOC, annotations, and complexity, this suggests that Test SATD tends to reside in larger and more complex test code. However, we also found that Test SATD is not associated with test smells (*i.e.,* they are independent). This implies that test smell detection tools are not identifying most of the test issues, which is a gap to be filled by researchers and tool developers. Compared with code smells, test smells are a relatively recent concept and are still under development. Therefore, we recommend that *researchers and tool developers should endeavor to identify new types of test smells, referring to Test SATD in practice.*

**Implication 3. Developers should consider using code completion tools specialized in test code.**

RQ3 categorizes the types of Test SATD. We found that the most frequent category was "Indicating incomplete or unimplemented tests." This suggests that developers often stop writing assertions and document unfinished tests or insufficient test cases as SATD. This behavior may be due to limited time or other constraints, but assertions are usually written with only a few lines of code. It is unclear why developers leave a few-line comments instead of writing assertions. Therefore,

we recommend that *researchers investigate why developers did not finish writing assertions through surveys or interviews.* Additionally, *practitioners could consider using copilot tools, especially code completion tools for test code [49, 78].*

**Implication 4. Test SATD can be automatically classified but the performance could be still improved.**

Existing SATD classification tools primarily focused on production code, and no classifier had been developed to categorize SATD in test code. To address this issue, in RQ4, we developed a classifier for Test SATD and trained it based on the classification results of RQ3. As a result, the classifier utilizing CodeBERT demonstrated the highest performance compared to other models. While the classifiers demonstrated the feasibility of automatically classifying Test SATD, they achieved a maximum F1-Score of 0.70, which is far from perfect. Therefore, *researchers and tool developers may perform further manual inspections to extend the dataset and retrain the model with a larger dataset to improve the classification performance.*

## 7 Threats to validity

*Threats to internal validity* concern the factors we did not consider that might impact the results. The manual classification in this study was independently conducted by two inspectors to reduce subjective bias. The Cohen's kappa coefficient was 0.78, indicating a high level of agreement. Disagreements were resolved with a third inspector. Despite these measures, the misinterpretation of SATD comments cannot be entirely ruled out.

*Threats to construct validity* concern the relation between theory and observation. This study analyzed SATD within test code. A previous study [7] has found that SATD instances in the production code may also point out test-related issues, which are not included in our studies. We believe the number of SATD instances related to testing in the production code is relatively small, but future research could consider production code when identifying test-related SATD.

The imbalanced distribution across categories poses validity concerns. The most infrequent category, "Failures," contains only 15 instances, which may not adequately represent the full spectrum of this construct. While this imbalance problem is common in software engineering research [60, 68], this imbalance directly affects the validity of our performance measurements: during 10-fold cross-validation, several test sets contained only a single "Failures" instance, potentially leading to unstable performance estimates that may not accurately reflect the model's true ability to identify this category. While using median performance metrics partially addresses this threat, we cannot claim equal measurement validity across all categories.

*Threats to external validity* concern the generalizability of our findings. In this study, the manual classification was performed on a sampled subset of detected SATD. To mitigate sampling bias, we applied random sampling to satisfy a 99% confidence level and a 5% margin of error, and the study included 50 repositories, which is more than in other studies [20, 36]. Nevertheless, the possible bias due to repository selection remains. Furthermore, our work primarily studied open-source projects because code access was necessary for the analysis. It is unclear whether proprietary projects exhibit the same patterns as shown in our results.

In addition, our sample of 506 instances may underrepresent rare but potentially important Test SATD categories (*e.g.,* our "Failures" category has only 15 instances). However, from a statistical perspective, the distribution of categories in our sample likely reflects their natural prevalence in software systems in the wild. The rarity of certain SATD types, such as "Failures," may indicate their actual low frequency of occurrence rather than a sampling bias. While increasing the dataset size might yield more instances of rare categories, the proportional distribution might remain similar.

## 8 Conclusion

In this study, we conduct a large-scale empirical investigation into the nature of Self-Admitted Technical Debt in test code, an area often overlooked in existing research. We collected 17,766 SATD comments (14,987 from production code, 2,779 from test code) from 50 repositories and analyzed their prevalence, relationship with various quality metrics, specific types, and the feasibility of their automatic classification.

Our findings reveal several key insights. First, Test SATD is a non-negligible phenomenon, accounting for 15.6% of all SATD instances across 50 projects. Second, we found that Test SATD correlates with general code quality issues like code smells and complexity, but notably, it does not correlate with test smells, suggesting that SATD and test smells represent distinct quality concerns. Third, through manual analysis, we developed a detailed taxonomy of 20 types of Test SATD, finding that the most common reason developers admit debt is due to incomplete or unimplemented tests. Finally, we demonstrated the feasibility of automatically classifying these SATD types, with a CodeBERT-based model showing the most balanced performance, though we also found that modern LLMs like GPT-4.1 can identify unique instances that other models miss.

Our future work will focus on two main directions: (i) expanding our manually labeled dataset to improve the performance and robustness of our classification models, and (ii) analyzing the evolution of Test SATD over time to understand its lifecycle and resolution patterns.

## Acknowledgments

We gratefully acknowledge the financial support of JSPS KAKENHI grants (JP24K02921, JP25K21359), as well as JST PRESTO grant (JPMJPR22P3), ASPIRE grant (JPMJAP2415), and AIP Accelerated Program (JPMJCR25U7).

## References

[1] Nicolli S. R. Alves, Manoel Gomes de Mendonça Neto, and Rodrigo Oliveira Spínola. 2018. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. *Information and Software Technology* 102 (2018), 117–145.

[2] Nicolli S. R. Alves, Thiago Souto Mendes, Manoel Gomes de Mendonça Neto, Rodrigo O. Spínola, Forrest Shull, and Carolyn B. Seaman. 2016. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology* 70 (2016), 100–121.

[3] Nicolli S. R. Alves, Leilane Ferreira Ribeiro, Vivyane Caires, Thiago Souto Mendes, and Rodrigo O. Spínola. 2014. Towards an Ontology of Terms on Technical Debt. In *Proceedings of the 6th International Workshop on Managing Technical Debt (MTD 2014)*. 1–7.

[4] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. 2014. Test Code Quality and Its Relation to Issue Handling Performance. *IEEE Transactions on Software Engineering* 40, 11 (2014), 1100–1125.

[5] Hideaki Azuma, Shinsuke Matsumoto, Yasutaka Kamei, and Shinji Kusumoto. 2022. An empirical study on self-admitted technical debt in Dockerfiles. *Empirical Software Engineering* 27, 2 (2022), 49.

[6] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave W. Binkley. 2015. Are test smells really harmful? An empirical study. *Empirical Software Engineering* 20, 4 (2015), 1052–1094.

[7] Gabriele Bavota and Barbara Russo. 2016. A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR 2016)*. 315–326.

[8] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32.

[9] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. 2022. Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques. In *Proceedings of the 19th IEEE/ACM International Conference on Mining Software Repositories (MSR 2022)*. 464–468.

[10] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. 2002. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research* 16 (2002), 321–357.

[11] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 785–794.

[12] Xin Chen, Dongjin Yu, Xulin Fan, Lin Wang, and Jie Chen. 2022. Multiclass Classification for Self-Admitted Technical Debt Based on XGBoost. *IEEE Transactions on Reliability* 71, 3 (2022), 1309–1324.

[13] Zadia Codabux and Byron J. Williams. 2013. Managing technical debt: an industrial case study. In *Proceedings of the 4th International Workshop on Managing Technical Debt (MTD 2013)*. 8–15.

[14] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. 2011. Lightweight Transformation and Fact Extraction with the srcML Toolkit. In *Proceedings of the 11th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM 2011)*. 173–184.

[15] Hugh Coolican. 2017. *Research methods and statistics in psychology.* Psychology press.

[16] Ward Cunningham. 1992. The WyCash portfolio management system. In *Addendum to the Proceedings on Object-Oriented Programming Systems*, Vol. 4. 29–30.

[17] Everton da S. Maldonado and Emad Shihab. 2015. Detecting and quantifying different types of self-admitted technical Debt. In *Proceedings of the 7th IEEE International Workshop on Managing Technical Debt (MTD 2015)*. IEEE Computer Society, 9–15.

[18] Everton da S. Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt. *IEEE Transactions on Software Engineering* 43, 11 (2017), 1044–1062.

[19] Mário André de Freitas Farias, Manoel Gomes de Mendonça Neto, André Batista da Silva, and Rodrigo Oliveira Spínola. 2015. A Contextualized Vocabulary Model for identifying technical debt on code comments. In *Proceedings of the 7th IEEE International Workshop on Managing Technical Debt (MTD 2015)*. 25–32.

[20] Mário André de Freitas Farias, Manoel Gomes de Mendonça Neto, Marcos Kalinowski, and Rodrigo Oliveira Spínola. 2020. Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. *Information and Software Technology* 121 (2020), 106270.

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT 2019)*. 4171–4186.

[22] Thomas G. Dietterich. 1998. Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms. *Neural Computation* 10, 7 (1998), 1895–1923.

[23] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: the developer's perspective. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE 2019)*. 830–840.

[24] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[25] Jernej Flisar and Vili Podgorelec. 2019. Identification of Self-Admitted Technical Debt Using Enhanced Feature Selection Based on Word Embedding. *IEEE Access* 7 (2019), 106475–106494.

[26] Martin Fowler. 1999. *Refactoring - Improving the Design of Existing Code.* Addison-Wesley.

[27] Vahid Garousi and Baris Küçük. 2018. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software* 138 (2018), 52–81.

[28] Israel Gat and John D. Heintz. 2011. From assessment to reduction: how cutter consortium helps rein in millions of dollars in technical debt. In *Proceedings of the 2nd Workshop on Managing Technical Debt (MTD 2011)*. 24–26.

[29] Robert J Grissom and John J Kim. 2005. *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers.

[30] Zhaoqiang Guo, Shiran Liu, Jinping Liu, Yanhui Li, Lin Chen, Hongmin Lu, Yuming Zhou, and Baowen Xu. 2019. MAT: A simple yet strong baseline for identifying self-admitted technical debt.

[31] Fatih Gurcan, Gonca Gokce Menekse Dalveren, Nergiz Ercil Cagiltay, Dumitru Roman, and Ahmet Soylu. 2022. Evolution of Software Testing Strategies and Trends: Semantic Content Analysis of Software Research Corpus of the Last 40 Years. *IEEE Access* 10 (2022), 106093–106109.

[32] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. 2018. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering* 23, 1 (2018), 418–451.

[33] Timo Hynninen, Jussi Kasurinen, Antti Knutas, and Ossi Taipale. 2018. Software testing: Survey of the industry practices. In *Proceedings of the 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 1449–1454.

[34] Karen Spärck Jones. 2004. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation* 60, 5 (2004), 493–502.

[35] Yasutaka Kamei, Everton Maldonado, Emad Shihab, and Naoyasu Ubayashi. 2016. Using analytics to quantify the interest of self-admitted technical debt. *CEUR Workshop Proceedings* 1771 (2016), 68–71.

[36] Yutaro Kashiwa, Ryoma Nishikawa, Yasutaka Kamei, Masanari Kondo, Emad Shihab, Ryosuke Sato, and Naoyasu Ubayashi. 2022. An empirical study on self-admitted technical debt in modern code review. *Information and Software Technology* 146 (2022), 106855.

[37] Yutaro Kashiwa, Kazuki Shimizu, Bin Lin, Gabriele Bavota, Michele Lanza, Yasutaka Kamei, and Naoyasu Ubayashi. 2021. Does Refactoring Break Tests and to What Extent?. In *Proceedings of the 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME 2021)*. 171–182.

[38] Igor Kononenko. 1990. Comparison of inductive and naive Bayesian learning approaches to automatic knowledge acquisition. *Current trends in knowledge acquisition* 8 (1990), 190.

[39] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. 2012. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software* 29, 6 (2012), 18–21.

[40] J Richard Landis and Gary G. Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* 33, 1 (1977), 159–174.

[41] Jun Li, Lixian Li, Jin Liu, Xiao Yu, Xiao Liu, and Jacky Wai Keung. 2025. Large language model ChatGPT versus small deep learning models for self-admitted technical debt detection: Why not together? *Softw. Pract. Exp.* 55, 1 (2025), 3–28.

[42] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A systematic mapping study on technical debt and its management. *Journal of Systems and Software* 101 (2015), 193–220.

[43] Zhongxin Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2018. SATD detector: a text-mining-based self-admitted technical debt detection tool. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings (ICSE 2018)*. 9–12.

[44] Rungroj Maipradit, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. 2020. Wait for it: identifying "On-Hold" self-admitted technical debt. *Empirical Software Engineering* 25, 5 (2020), 3770–3798.

[45] Salvatore S Mangiafico. 2016. Summary and analysis of extension program evaluation in R. https://rcompanion.org/handbook/ (Accessed: 2025-02-01).

[46] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[47] Luana Almeida Martins, Heitor A. X. Costa, Márcio Ribeiro, Fabio Palomba, and Ivan Machado. 2023. Automating Test-Specific Refactoring Mining: A Mixed-Method Investigation. In *Proceedings of the 23rd IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2023)*. 13–24.

[48] Steve McConnell. 2013. Managing Technical Debt. *Construx* (2013), 1–14.

[49] Pengyu Nie, Rahul Banerjee, Junyi Jessy Li, Raymond J. Mooney, and Milos Gligoric. 2023. Learning Deep Semantics for Test Completion. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE 2023)*. 2111–2123.

[50] Ariadi Nugroho, Joost Visser, and Tobias Kuipers. 2011. An empirical model of technical debt and interest. In *Proceedings of the 2nd Workshop on Managing Technical Debt (MTD 2011)*. 1–8.

[51] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. 2017. An exploratory study on the relationship between changes and refactoring. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC 2017)*. 176–185.

[52] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. tsDetect: an open source test smells detection tool. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. 1650–1654.

[53] Aniket Potdar and Emad Shihab. 2014. An Exploratory Study on Self-Admitted Technical Debt. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME 2014)*. 91–100.

[54] Leevi Rantala, Mika Mäntylä, and Valentina Lenarduzzi. 2024. Keyword-labeled self-admitted technical debt and static code analysis have significant relationship but limited overlap. *Software Quality Journal* 32, 2 (2024), 391–429.

[55] Leevi Rantala, Mika Mäntylä, and David Lo. 2020. Prevalence, Contents and Automatic Detection of KL-SATD. In *Proceedings of the 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2020)*. 385–388.

[56] Xiaoxue Ren, Zhenchang Xing, Xin Xia, David Lo, Xinyu Wang, and John Grundy. 2019. Neural Network-based Detection of Self-Admitted Technical Debt: From Performance to Explainability. *ACM Transactions on Software Engineering and Methodology* 28, 3 (2019), 15.

[57] Ahmed F. Sabbah and Abualsoud A. Hanani. 2023. Self-admitted technical debt classification using natural language processing word embeddings. *International Journal of Electrical and Computer Engineering (IJECE)* 13, 2 (2023), 2142–2155.

[58] Irene Sala, Antonela Tommasel, and Francesca Arcelli Fontana. 2021. DebtHunter: A Machine Learning-based Approach for Detecting Self-Admitted Technical Debt. In *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering (EASE 2021)*. 278–283.

[59] Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. 2018. A comprehensive model for code readability. *J. Softw. Evol. Process.* 30, 6 (2018).

[60] Chris Seiffert, Taghi M. Khoshgoftaar, Jason Van Hulse, and Andres Folleco. 2014. An empirical study of the classification performance of learners on imbalanced and noisy software quality data. *Inf. Sci.* 259 (2014), 571–595.

[61] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2017. House of Cards: Code Smells in Open-Source C# Repositories. In *Proceedings of the 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2017)*. 424–429.
[62] Mohammad Sadegh Sheikhaei, Yuan Tian, Shaowei Wang, and Bowen Xu. 2024. An empirical study on the effectiveness of large language models for SATD identification and classification. *Empirical Software Engineering* 29, 6 (2024), 159.
[63] Giancarlo Sierra, Emad Shihab, and Yasutaka Kamei. 2019. A survey of self-admitted technical debt. *Journal of Systems and Software* 152 (2019), 70–82.
[64] Elvys Soares, Márcio Ribeiro, Rohit Gheyi, Guilherme Amaral, and André L. M. Santos. 2023. Refactoring Test Smells With JUnit 5: Why Should Developers Keep Up-to-Date? *IEEE Transactions on Software Engineering* 49, 3 (2023), 1152–1170.
[65] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. 2018. On the Relation of Test Smells to Software Code Quality. In *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME 2018)*. 1–12.
[66] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.
[67] Ronald J Tallarida, Rodney B Murray, Ronald J Tallarida, and Rodney B Murray. 1987. Chi-square test. *Manual of pharmacologic calculations: with computer programs* (1987), 140–142.
[68] Chakkrit Tantithamthavorn, Ahmed E. Hassan, and Kenichi Matsumoto. 2020. The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models. *IEEE Trans. Software Eng.* 46, 11 (2020), 1200–1219.
[69] Igor V. Tetko, David J. Livingstone, and Alexander I. Luik. 1995. Neural network studies, 1. Comparison of overfitting and overtraining. *J. Chem. Inf. Comput. Sci.* 35, 5 (1995), 826–833.
[70] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. 4–15.
[71] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. 2001. Refactoring Test Code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering*. 92–95.
[72] V. Vapnik. 1963. Pattern recognition using generalized portrait method. *Automation and Remote Control* 24 (1963), 774–780.
[73] Carmine Vassallo, Fiorella Zampetti, Daniele Romano, Moritz Beller, Annibale Panichella, Massimiliano Di Penta, and Andy Zaidman. 2016. Continuous Delivery Practices in a Large Financial Organization. In *Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME 2016)*. 519–528.
[74] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. 2016. Examining the Impact of Self-Admitted Technical Debt on Software Quality. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution and Reengineering (SANER 2016)*. 179–188.
[75] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83.
[76] Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. 2007. On Early Stopping in Gradient Descent Learning. *Constructive Approximation* 26 (2007), 289–315.
[77] Nico Zazworka, Michele A. Shaw, Forrest Shull, and Carolyn B. Seaman. 2011. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt (MTD 2011)*. 17–23.
[78] Tingwei Zhu, Zhongxin Liu, Tongtong Xu, Ze Tang, Tian Zhang, Minxue Pan, and Xin Xia. 2024. Exploring and Improving Code Completion for Test Code. In *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension (ICPC 2024)*. 137–148.

Received 10 February 2025; revised 26 August 2025; accepted 8 December 2025