

Is Self-Admitted Technical Debt Tested?

An Empirical Study of Coverage, Co-Change, and Impact

Suzuka Yoshimoto 

Nara Institute of Science and Technology, Japan

Kosei Horikawa 

Nara Institute of Science and Technology, Japan

Daniel Feitosa 

University of Groningen, the Netherlands

Yutaro Kashiwa  

Nara Institute of Science and Technology, Japan

Hajimu Iida 

Nara Institute of Science and Technology, Japan

Abstract

Background. When developers write a `TODO` or `FIXME` comment, they are explicitly admitting that the code is suboptimal: a built-in warning that this logic deserves extra scrutiny. Yet it is an open question whether Self-Admitted Technical Debt (SATD) actually receives that scrutiny in the form of software testing.

Aim. We aim to characterize the relationship between SATD and testing across three dimensions: the extent to which SATD-affected code is covered by existing tests, whether developers synchronize test additions with debt resolution, and whether such testing affects the long-term observability of resulting defects.

Method. For that, we conducted an empirical study on eight open-source Java projects, analyzing test coverage of 784 SATD instances identified in the latest releases and performing a longitudinal examination of 5,175 SATD removal events.

Results. Our results show that while 60.7% of SATD-affected code is covered by existing test suites, developers rarely synchronize test modifications with debt resolution; manual inspection confirms that only 3.4% of SATD removal commits include new tests specifically targeting the resolved debt (vs. 12.5% that co-add tests in the same commit). Longitudinal analysis further suggests that SATD resolutions exhibit nearly identical localized bug induction rates within short-to-medium-term windows regardless of test modifications. However, over a longer, unrestricted observation window, a slight divergence emerges where the test-added group reaches a higher cumulative defect alignment probability (6.32% vs. 4.37%), a counterintuitive trend potentially driven by the selective testing of inherently complex components.

Conclusion. Developers treat SATD repayment as an ordinary code change rather than as a high-risk maintenance activity: most debt removals proceed without targeted verification, despite the developer's own prior flag that the code is suboptimal.

2012 ACM Subject Classification Software and its engineering; Software and its engineering → Risk management

Keywords and phrases Self-Admitted Technical Debt, Test Coverage, Software Testing

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.36

Category Technical Track Paper



© Suzuka Yoshimoto, Kosei Horikawa, Daniel Feitosa, Yutaro Kashiwa, and Hajimu Iida; licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón; Article No. 36; pp. 36:1–36:20



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Funding We gratefully acknowledge the financial support of JSPS KAKENHI grants (JP24K02921, JP25K03100, JP26K23818, JP26H02500), as well as ASPIRE grant (JPMJAP2415), JST BOOST grant (JPMJBS2423), JST CREST grant (JPMJCR23M1, JPMJCR26X7), and JST FOREST (JPMJFR252W).

1 Introduction

Technical debt is an inherent byproduct of software development: facing deadlines or incomplete specifications, developers sometimes implement solutions they know are suboptimal [24]. Rather than leaving that knowledge implicit, they often record it directly in the source code through comments such as `TODO`, `FIXME`, and `HACK`. This practice, known as Self-Admitted Technical Debt (SATD) [29], is surprisingly widespread: between 2.4% and 31% of source files in open-source projects contain such comments [29], and the flagged code tends to persist for tens to hundreds of days before being resolved [30, 9], during which time it continues to be more change-prone and defect-prone than surrounding code [36].

What makes SATD distinctive is that the developer’s own comment serves as an explicit warning: this code is known to be fragile, incomplete, or incorrect. Unlike latent defects that remain hidden until discovered, SATD represents a conscious acknowledgment that the implementation may not behave as intended. Such code should therefore be subject to verification, and software testing is the primary mechanism through which developers confirm that code behaves correctly and does not regress under future changes.

Software testing serves as a primary mechanism for early defect detection and regression prevention. A rich body of literature has examined the relationship between test coverage and defect detection capability [19], revealing, for example, that higher statement or branch coverage correlates with greater fault-detection effectiveness, though the strength of this relationship varies across projects and coverage metrics [21]. Researchers have also studied how test modifications affect code quality, finding that commits accompanied by test changes tend to introduce fewer regressions than those without [35]. More broadly, tests act as a safety net during refactoring and code modifications, and the presence or absence of adequate tests can influence the success of such changes [39].

Despite these parallel lines of research, SATD studies and testing studies have progressed largely in isolation. These findings are naturally complementary, yet the relationship between SATD and testing remains largely unexplored. First, it is unknown to what extent code containing SATD is protected by test suites. Second, there is a lack of empirical evidence on how often commits that remove SATD are accompanied by changes to test code, that is, the extent to which developers are conscious of testing during the SATD lifecycle. Third, whether adding tests at the time of SATD removal actually reduces the rate of subsequent bug occurrences has not been verified.

To address these gaps, we conducted an empirical study on eight open-source Java projects hosted on GitHub, examining the relationship between SATD and testing across two temporal scales. First, we identified 784 SATD instances in the latest releases to evaluate their current test coverage. Second, we performed a longitudinal analysis of 5,175 SATD removal events to investigate the co-occurrence of test modifications and the long-term observability of resulting defects. Based on this extensive dataset, we pose the following three research questions and summarize our key findings:

RQ₁: To what extent is SATD covered by tests compared to the overall codebase?

We found that 60.7% of SATD-affected code is covered by test cases. Furthermore, when categorizing SATD into technical types (e.g., Design, Requirement, and Defect debt), median coverage remained similar across types (41.7%–50.0%), indicating that the specific type of

debt does not significantly influence whether it gets tested. These results indicate that a majority of admitted debt is within the reach of existing verification frameworks. Notably, in several projects, SATD-affected code exhibited higher test coverage compared to the overall codebase, indicating that developers do not necessarily neglect brittle areas.

RQ₂: To what extent do developers add targeted tests when removing SATD?

Despite the high baseline coverage, our analysis of 5,175 removed SATD revealed a disconnect during the debt lifecycle; only 3.4% of SATD removals included concurrent modifications to test code. This suggests that while developers may rely on a pre-existing safety net, they rarely update or tailor tests specifically in response to the resolution of known debt.

RQ₃: What is the impact of removing SATD without adequate testing on software quality?

To understand the impact of these rare testing efforts, we analyzed the relationship between bug induction and SATD removals that included concurrent test additions. Our results indicate that the presence of test modifications yields no immediate reduction in localized bug induction rates within short-to-medium-term windows (*e.g.*, 30, 90, and 180 days). Interestingly, over an unrestricted time window, a slight divergence becomes visible, with the test-added group exhibiting a higher cumulative probability of subsequent defect alignment than the no-test group. This counterintuitive pattern suggests that general test modifications within a commit do not inherently prevent regressions, potentially because they lack sufficient coverage for the resolved logic. Alternatively, this may reflect a selection bias where developers selectively implement tests for particularly complex or critical SATD instances that remain inherently prone to defects.

2 Motivating Example

Software maintenance requires a continuous, synchronized effort between the evolution of production code and its corresponding test suite. Prior studies on code co-evolution, such as the work by *Levin and Yehudai* [23], have observed that this synchronization is often neglected in practice; developers frequently perform code fixes without correspondingly updating or adding tests within the same commit. While such findings highlight a general disconnect in maintenance activities, a critical question remains: *Does this trend persist even when developers have explicitly flagged the code as fragile or suboptimal?*

When developers write an SATD comment like `TODO` or `FIXME`, they are explicitly admitting that the logic is suboptimal – a built-in warning that the code deserves extra scrutiny. In an ideal development workflow, the resolution of such debt should be a controlled, verified process. For instance, in the Dubbo project (Listing 1), a developer identified a limitation where the system only checked a single registry. When this debt was eventually repaid (Listing 2), the developer not only refactored the logic but also provided a corresponding test case (Listing 3) to verify the new implementation. This scenario illustrates a rigorous approach where the removal of admitted debt is immediately fortified by quality assurance.

However, the presence of a `TODO` or `FIXME` does not always act as a trigger for such caution. Another scenario is illustrated by the Hibernate project. In Listing 4, a developer acknowledged a “wierd implementation” caused by “hacky behavior.” While the SATD comment was later removed in an attempt to clean up the logic (Listing 5), this change was not accompanied by any new tests. Without a safety net, this testless debt removal introduced a regression that was later identified as a bug-inducing change.

This discrepancy reveals a blind spot in technical debt management. While prior research has focused heavily on the identification and classification of SATD, there is a lack of systematic understanding regarding the testing. Even if general code changes often lack

36:4 Is Self-Admitted Technical Debt Tested?

■ **Listing 1** Original code with SATD (DUBBO, ServiceCheckUtils.java)

```
1 public static boolean isRegistered(ProviderModel providerModel) {
2     // TODO, only check the status of one registry and no protocol now.
3     Collection<Registry> registries = registryManager.getRegistries(); // setup
        omitted
4     if (CollectionUtils.isNotEmpty(registries)) {
5         AbstractRegistry abstractRegistry = (AbstractRegistry) registries.
            iterator().next();
6         // ... omitted for brevity ...
7     }
8 }
```

■ **Listing 2** Refactored code after SATD removal (DUBBO, ServiceCheckUtils.java)

```
1 public static boolean isRegistered(ProviderModel providerModel) {
2     // check all registries status
3     for (ProviderModel.RegisterStatedURL registerStatedURL :
        providerModel.getStatedUrl()) {
4         if (registerStatedURL.isRegistered()) {
5             return true;
6         }
7     }
8 }
```

■ **Listing 3** Newly added test code accompanying the SATD removal (DUBBO, ServiceCheckUtilsTest.java)

```
1 @Test
2 public void testIsRegistered() {
3     // ... test setup omitted for brevity ...
4     boolean registered = ServiceCheckUtils.isRegistered(providerModel);
5     assertFalse(registered);
6 }
```

■ **Listing 4** Original code with SATD (HIBERNATE, DenormalizedTable.java)

```
1 @Override
2 public Iterator getUniqueKeyIterator() {
3     //wierd implementation because of hacky behavior
4     //of Table.sqlCreateString() which modifies the
5     //list of unique keys by side-effect on some
6     //dialects
7     Map uks = new HashMap();
8     uks.putAll( getUniqueKeys() );
9     uks.putAll( includedTable.getUniqueKeys() );
10    return uks.values().iterator();
11 }
```

tests, the failure to verify code that has been explicitly admitted as problematic represents a higher-risk category of neglect. This study aims to bridge this gap by quantifying the relationship between testing practices and the SATD lifecycle, providing the empirical link between these two previously disconnected domains of software engineering research.

■ **Listing 5** Bug-inducing code: SATD removed without accompanying test additions (HIBERNATE, `DenormalizedTable.java`)

```

1 @Override
2 public Iterator getUniqueKeyIterator() {
3     Iterator iter = includedTable.getUniqueKeyIterator();
4     while ( iter.hasNext() ) {
5         UniqueKey uk = (UniqueKey) iter.next();
6         createUniqueKey( uk.getColumns() );
7     }
8     return getUniqueKeys().values().iterator();
9 }

```

3 Study Design

3.1 Research Questions

The following research questions and their motivations guide our empirical investigation.

***RQ₁*: To what extent is SATD covered by tests compared to the overall codebase?**

SATD represents a developer’s own acknowledgment that a piece of code is suboptimal or incomplete. Prior work has shown that such debt-ridden code tends to be more change-prone and defect-prone than surrounding code [36], and that SATD comments persist for tens to hundreds of days before being resolved [30, 9]. Given these known risks, one might expect developers to compensate with more thorough testing. Yet it is equally plausible that writing tests for SATD-affected code, much like the resolution of the debt itself, is also postponed under time pressure. While prior studies have examined the relationship between test coverage and defect detection [19, 21], no study has directly investigated whether SATD-affected code receives adequate test coverage. We therefore aim to determine the extent to which SATD is covered by test suites and how this coverage varies across different types of debt.

***RQ₂*: To what extent do developers add targeted tests when removing SATD?**

Static coverage alone cannot tell us whether developers are consciously investing in test quality during the SATD lifecycle, or whether any existing coverage is merely incidental. Prior studies on developer behavior during refactoring have found that test updates are often neglected even when production code changes substantially [39]. Whether this pattern extends to SATD-related commits, where the code is already known to be suboptimal, remains an open question. Understanding this reveals whether quality assurance is treated as an integral part of debt management or whether it is systematically overlooked during these high-risk changes.

***RQ₃*: What is the impact of removing SATD without adequate testing on software quality?**

Prior work has shown that SATD-related changes tend to be more complex and harder to perform than typical modifications [36], indicating that debt resolution carries inherent risks. However, no study has directly quantified how concurrent test additions during SATD removal influence the observability of subsequent defects. Therefore, we aim to evaluate the effect of concurrent test additions on linked bug-inducing rates during debt resolution, investigating whether such tests serve as an effective diagnostic mechanism for capturing regressions that would otherwise persist as undetected failures.

■ **Table 1** Overview of SATD found within method bodies in analyzed projects.

Repository	Release	Module(s)	#Comments	#SATD	% SATD
<i>Group 1: Projects with preconfigured Jacoco plugin</i>					
COMMONS LANG [4]	commons-lang-3.18.0	src	1,042	28	2.7
COMMONS IO [3]	commons-io-2.20.0	src	568	23	4.0
HIBERNATE [15]	6.6.28	hibernate-core	11,808	530	4.5
DUBBO [5]	dubbo-3.3.5	dubbo-common	924	21	2.3
<i>Group 2: Projects where Jacoco plugin was manually added</i>					
JFREECHART [20]	v1.5.6	src	2,865	60	2.1
SPOON [17]	v11.2.1	src	2,207	25	1.1
MAVEN [6]	maven-3.9.11	maven-core, maven-compact	862	69	8.0
STORM [7]	v2.8.2	storm-server, storm-client	3,563	43	1.2
Total			23,839	799	3.4

3.2 Subject Projects

For our study, we selected a diverse set of eight open-source Java projects, as detailed in Table 1. While purely static SATD studies analyze larger corpora [28, 27], our design prioritizes dynamic test execution and manual inspection, requiring stable build environments and a deliberate trade-off in corpus size. The chosen projects vary in domain and scale, covering utility libraries, application frameworks, specialized tools, and build infrastructure. This allows us to generalize our findings across different software development contexts. For each project, we analyzed a specific, recent release to ensure the relevance of our findings. The projects are divided into two categories based on their pre-existing test coverage infrastructure, which is central to our analysis.

Jacoco-preconfigured projects comprises four projects that already had the Jacoco test coverage plugin configured. This group includes foundational core utility libraries (COMMONS LANG and COMMONS IO) and large-scale application frameworks (HIBERNATE, an Object-Relational Mapping framework, and DUBBO, a microservice framework).

Manually configured projects comprises four projects for which we manually added the Jacoco plugin to perform our analysis. This group includes specialized libraries (JFREECHART, a charting library, and SPOON, a metaprogramming library) and development infrastructure tools (MAVEN, a build automation tool, and STORM, a real-time stream processing system).

3.3 Data Collection and Processing Pipeline

Our methodology involves a multi-stage pipeline to detect SATD, measure its test coverage, and trace its lifecycle and impact on defects.

1) SATD Detection and Classification. We employed DebtHunter [31], a state-of-the-art, ML-based SATD detection tool. The study introducing the tool demonstrated its high performance, with a precision of 97.2%, a recall of 96.7%, and an F1-score of 96.5%, outperforming other detection tools. In addition to SATD detection, we use DebtHunter to classify SATD by type. Since our study focuses on technical debt within executable logic, we modified the tool for two purposes: (1) to restrict detection to comments within method bodies, and (2) to extract the location of each SATD instance, including file name, method name, and line number.

■ **Table 2** Classification of all 799 SATD instances.

Type	#	%
Design	518	64.8
Requirement	256	32.0
Defect	22	2.8
Test	3	0.4
Documentation	0	0.0

To independently verify the reliability of DebtHunter, we manually validated its detection results on the three projects not included in its original training: DUBBO, SPOON, and STORM. We inspected all 129 comments flagged as SATD by the tool across these three projects (prior to applying the method body restriction). Our review confirmed that the classifications for SPOON and STORM were highly accurate, identifying zero false positives. For DUBBO, however, we found 10 false positives, which were non-SATD comments related to configuration (*e.g.*, `load dubbo.applications.xxx`). We excluded these 10 false positives from our dataset before proceeding with the analysis.

Using DebtHunter, we categorized the instances into five types: Design, Requirement, Defect, Test, and Documentation. As shown in Table 2, our dataset is dominated by Design (logical structure) and Requirement (requirement specifications) debt.

2) Measuring Test Coverage via Tracer Lines. To measure test coverage, we transform the passive SATD comment into an active, executable tracer line. For each identified SATD, we attempt to replace the comment line with an executable statement (*i.e.*, `System.out.println("SATD: IMPLEMENTATION");`). We then run the project's entire test suite and analyze the generated jacoco.xml reports. An SATD is considered covered if an inserted tracer line is reported as executed by Jacoco. However, this direct replacement fails in two scenarios, for which we manually analyze the SATD and map it to an appropriate executable line within the relevant debt-ridden code block.

(i) *Inappropriate Location:* The comment is not located within the actual debt-ridden code block it refers to. For example, in Listing 6, the `FIXME` comment is placed above the `if` block, but it refers to the logic within the `for` loop. Replacing the comment line directly would test the area outside the relevant logic.

■ **Listing 6** Example of an SATD comment in an inappropriate location (HIBERNATE, `MapBinder.java`).

```

1 //FIXME pass the Index Entity JoinColumns
2 if ( !collection.isOneToMany() ) {
3     //index column should not be null
4     for ( AnnotatedJoinColumn column : mapKeyManyToManyColumns.getJoinColumns()
5         ) {
6         column.forceNotNull();
7     }
8 }

```

(ii) *Syntax Error:* The replacement violates Java syntax. In Listing 7, the `TODO` comment is located within a variable declaration statement. Inserting an executable statement at the comment's location would break the assignment logic, causing a compilation error.

■ **Listing 7** Example of an SATD comment causing a syntax error if replaced (HIBERNATE, `SqmSelectionQueryImpl.java`).

```

1 protected List<R> doList() {
2     final boolean containsCollectionFetches =
3         //TODO: why is this different from QuerySqmImpl.doList()?
4         statement.containsCollectionFetches();
5     // ... omitted for brevity ...
6 }

```

3) Mining SATD Lifecycle and Test Additions. To detect SATD-removing events, we utilized a customized version of SATDBailiff [1], a specialized framework for mining SATD history. We replaced its original SATD detecting engine with DebtHunter to ensure methodological consistency. By applying this modified version of SATDBailiff to the commit history of our target projects, we were able to systematically identify commits where SATD was removed. We then analyzed these commits to determine if they included any modifications to the test suite.

4) Linking SATD Removal to Defects. To investigate the impact of SATD removal on software quality, we analyzed the relationship between SATD removal events and subsequent bug introductions. We first identified bug-fixing commits across the eight Java projects and employed LLM4SZZ [33], an advanced SZZ variant, to identify the specific commits and methods that introduced the bugs.

Unlike commit-level analysis, we performed a method-level matching to ensure high precision. Specifically, we identified instances where a bug was introduced within the same method from which the SATD had been removed. This allowed us to correlate the act of debt resolution directly with the emergence of subsequent defects. Finally, we compared the linked bug-inducing rates between SATD removals accompanied by test additions and those that were not. Because SZZ can only identify bugs that were eventually detected and fixed, this rate captures defects that were observable enough to surface in the project’s bug-fixing history. This comparison examines whether adding tests helps surface latent issues in these fragile areas rather than leaving them as undetected regressions.

3.4 Data Analysis Methodology

Our analysis follows a two-fold approach. First, we perform a cross-sectional analysis of the latest releases to measure test coverage of SATD (RQ1). We compare SATD coverage against project-level branch coverage baselines. We then perform a longitudinal analysis of the projects’ history (RQ2 and RQ3). We investigate the frequency of test additions during SATD removals (RQ2), and compare the bug introduction rates between SATD removals accompanied by tests against those that were not (RQ3), quantifying the risks associated with testless debt repayment.

4 Results

***RQ₁*: To what extent is SATD covered by tests compared to the overall codebase?**

We applied our tracer-line instrumentation approach to all 799 SATD instances. Table 3 presents the detailed breakdown of instrumentation process and the final test coverage results. First, we attempted to automatically replace the comment with an executable tracer.

Instances that compiled successfully and were deemed to be in a suitable location were categorized as Automatic (B). The remaining instances either failed the automatic step (e.g., Syntax Error) or were manually identified as being in an Inappropriate Location. We attempted to manually map these instances to a suitable executable line. The instances that were successfully mapped constitute the Manual (C) category. Instances that could not be mapped were Excluded (D), primarily because the SATD comment was located within a code block that was already commented out. Consequently, the summation of the successfully instrumented instances from the Automatic and Manual categories constitutes the final set of Analyzed SATD (E) (i.e., $E = B + C$).

■ **Table 3** SATD Instrumentation Feasibility and Test Coverage Results per Project, Compared to Overall Project Branch Coverage.

Project	Filtering Process					Coverage Result (%)	
	All SATD (A)	Auto. (B)	Man. (C)	Excl. (D)	Anal. (E)	SATD Cov. (%)	Overall Branch Cov. (%)
<i>Group 1: Projects with pre-configured Jacoco plugin</i>							
COMMONS LANG	28	6	20	2	26	26.9*	92.9
COMMONS IO	23	22	0	1	22	95.5	85.7
HIBERNATE	530	463	65	2	528	70.8	58.5
DUBBO	21	11	0	10	11	45.5	53.7
<i>Group 2: Projects where Jacoco plugin was manually added</i>							
JFREECHART	60	58	2	0	60	40.0	46.9
SPOON	25	21	4	0	25	52.0	55.8
MAVEN	69	55	14	0	69	30.4	32.0
STORM	43	39	4	0	43	25.6	11.9
Total	799	675	109	15	784	60.7	—

* This low coverage is an outlier caused by a cluster of 18 identical, untested Design SATD in Validate.java.

The coverage results in Table 3 provide the first answer to our research question. Overall, we found that 60.7% (476 out of 784) of the analyzable SATD instances are covered by tests. A deeper analysis reveals that the projects are not uniform and instead follow three distinct testing patterns.

1. *Extra Care*: In three projects – COMMONS IO, HIBERNATE (both Group 1), and STORM (Group 2) – we observed SATD coverage that is higher than the project’s overall branch coverage baseline (e.g., COMMONS IO: 95.5% vs. 85.7% baseline; STORM: 25.6% vs. 11.9% baseline). This suggests that in these projects, developers pay at least equal, if not more, attention to known debt.

2. *General Postponement*: In four projects – DUBBO (from Group 1), and JFREECHART, SPOON, and MAVEN (from Group 2) – the SATD coverage was roughly proportional to, or slightly lower than, their project baselines (e.g., DUBBO: 45.5% vs. 53.7% baseline; MAVEN: 30.4% vs. 32.0% baseline). The DUBBO case is particularly revealing: despite being in the Jacoco-configured group, its SATD is not prioritized, indicating that a pre-existing testing culture does not, by itself, guarantee extra care for SATD.

3. *Targeted Postponement*: COMMONS LANG (from Group 1) presents a notable contrast. Despite maintaining a strict CI threshold that yields a high observed branch coverage of 92.9%, its SATD coverage is remarkably low at 26.9%.

We investigated this outlier and found it is caused by a single cluster of 18 identical Design SATD comments in the Validate.java file (see Listing 8). The comment refers to a potential future enhancement changing a `void` method to return a `value` on the success path.

36:10 Is Self-Admitted Technical Debt Tested?

■ **Table 4** Distribution of Test Coverage rates per SATD Type and Pattern.

Classification	1. Extra Care (Tested / Total)	2. General Post. (Tested / Total)	3. Targeted Post. (Tested / Total)
Design	243/ 365 (66.6%)	48/ 116 (41.4%)	6/ 24 (25.0%)
Requirement	144/ 204 (70.6%)	15/ 48 (31.3%)	1/ 2 (50.0%)
Defect	16/ 21 (76.2%)	0/ 1 (0.0%)	–
Test	3/ 3 (100.0%)	–	–
Total	406 / 593 (68.5%)	63 / 165 (38.2%)	7 / 26 (26.9%)

Our instrumentation (Listing 9) confirmed that this success path is entirely untested. The project’s high 92.9% branch coverage is achieved by thoroughly testing only the failure path (the `IllegalArgumentException`). This demonstrates that even strictly-enforced project-metric thresholds can be insufficient, as developers may focus tests on satisfying CI while leaving known technical debt on other paths untested.

■ **Listing 8** Example of 18 identical SATD comments in their original, inappropriate location (COMMONS LANG, `Validate.java`).

```
1 public static void exclusiveBetween(final double start, final double end, final
   double value) {
2     // TODO when breaking BC, consider returning value
3     if (value <= start || value >= end) {
4         throw new IllegalArgumentException(String.format(
5             DEFAULT_EXCLUSIVE_BETWEEN_EX_MESSAGE, value, start, end));
6     }
7 }
```

■ **Listing 9** The code from Listing 8 after manual instrumentation with the tracer line.

```
1 public static void exclusiveBetween(final double start, final double end, final
   double value) {
2     if (value <= start || value >= end) {
3         throw new IllegalArgumentException(String.format(
4             DEFAULT_EXCLUSIVE_BETWEEN_EX_MESSAGE, value, start, end));
5     }
6     System.out.println("SATD:␣DESIGN");
7 }
```

Coverage by SATD Type. Beyond project-level patterns, we investigated whether the type of technical debt influences its likelihood of being tested. Table 4 summarizes the coverage rates for each SATD type across the three project patterns. Within each pattern, the differences among SATD types are modest compared to the differences between patterns: in the Extra Care group, Design (66.6%), Requirement (70.6%), and Defect (76.2%) all attain coverage above 66%, while in the General Postponement group, coverage drops to 41.4% for Design and 31.3% for Requirement with the single Defect instance left uncovered. The Targeted Postponement group records the lowest values (Design: 25.0%, Requirement: 50.0%).

Aggregating across projects yields the same picture. The median coverages for Design (50.0%), Requirement (41.7%), and Defect (50.0%) cluster closely together with widely overlapping interquartile ranges. Contrary to the intuitive expectation that high-risk defect debt would be prioritized for testing, the technical type of SATD does not appear to influence whether it gets tested.

Test debt is a special case: all 3 instances come from HIBERNATE and yield a median of 100%. These comments flag inadequacies in the existing tests themselves, and the surrounding code was fully exercised even though the developer considered the test logic incomplete.

Answer to RQ1

We found that 60.7% (476 out of 784) of the analyzable SATD instances are covered by tests, and there are three distinct patterns of care: (1) Extra Care in three projects, (2) General Postponement in four projects, and (3) Targeted Postponement in Commons Lang where debt-ridden paths were selectively ignored.

Furthermore, all three major SATD types (Design, Requirement, and Defect) show similar median coverages (clustered between 41.7% – 50.0%) and widely overlapping distributions, suggesting that the technical type of SATD does not significantly influence whether it gets tested.

RQ₂: To what extent do developers add targeted tests when removing SATD?

Findings from RQ1 revealed that over 60% of SATD instances are covered by existing tests in latest release revision. This suggests that code containing technical debt is generally within the reach of the project’s testing infrastructure. However, from a software maintenance perspective, a critical juncture is the resolution of this debt. When an SATD comment is removed, it often signifies that the suboptimal logic has been refactored or replaced. It is therefore vital to understand whether such SATD removals trigger the creation of new tests to verify the updated implementation.

To identify commits involving SATD removals, we applied a modified version of SAT-DBailiff [1] to the main branch of each project. While the original tool relies on SATD Detector, we integrated DebtHunter as the underlying engine to maintain consistency with our identification criteria in RQ1 (see Table 5 for per-project counts).

Our methodology for detecting test additions involves a logical diff of test method names between consecutive commits. First, we perform AST analysis (using `javalang`¹) on each file in the main branch to extract sets of method names marked with the `@Test` annotation. By comparing these sets across revisions, we identify newly added test methods. To prevent false positives caused by simple refactorings, such as method renaming, we cross-reference Git patch information to verify the similarity between added and deleted lines. Test methods identified as renames are excluded, ensuring that we only count genuine functional additions to the test suite.

In this analysis, we focus on simultaneous co-evolution within a single commit. This is supported by Wang *et al.* [35], who found that production and test code co-evolution primarily happens in the same commit (median $\approx 47.0\%$) and that the number of co-evolutions decreases over time. Thus, same-commit analysis serves as a reliable proxy for determining

¹ <https://github.com/c2nes/javalang>

■ **Table 5** Test Addition Rates for SATD Removal Instances.

Project	Total SATD	w/ Co-added Tests (%)	w/ Verified Tests (%)
<i>Extra Care (RQ1 pattern)</i>			
Hibernate	3125	439(14.0%)	123(3.9%)
Storm	200	7(3.5%)	1(0.5%)
Commons IO	22	1(4.5%)	1(4.5%)
<i>General Postponement (RQ1 pattern)</i>			
Maven	1188	37(3.1%)	6(0.5%)
Dubbo	323	127(39.3%)	26(8.0%)
JFreeChart	173	1(0.6%)	1(0.6%)
Spoon	67	34(50.7%)	13(19.4%)
<i>Targeted Postponement (RQ1 pattern)</i>			
Commons Lang	77	3(3.9%)	3(3.9%)
Total	5175	649(12.5%)	174(3.4%)

Note: **Co-added Tests:** Tests introduced in the same commit as the SATD removal instance.
Verified Tests: A subset of co-added tests manually confirmed to specifically target the logic of the removed SATD.

whether developers add targeted tests as part of the SATD removal process. To ensure the validity of our findings, we complemented the automated identification with a rigorous manual inspection. While we first identified 649 SATD instances where test addition and SATD removal occurred within the same commit, these automated results do not necessarily imply a causal or logical relationship between the two.

To bridge this gap, two of the authors independently inspected each of the 649 instances to determine whether the added test methods were specifically intended to exercise or verify the logic related to the removed SATD. For each instance, both inspectors examined the code changes in the commit and the content of the newly added test methods to assess their relevance to the resolved debt. The two inspectors initially agreed on the classification for 78.3% of the instances, yielding a Cohen’s κ of 0.55, which indicates moderate agreement according to the interpretation by Landis and Koch [22]. Additionally, during this initial screening, the inspectors identified and filtered out 127 false positives produced by DebtHunter (e.g., non-SATD comments incorrectly flagged as SATD) to ensure data purity. Disagreements were subsequently resolved through adjudication by a third author, who independently reviewed the conflicting cases and made the final determination. This dual-inspection protocol with third-party adjudication allows for a more precise and reproducible assessment of how often developers explicitly address technical debt through testing. The inspectors have over 5-15 years of programming experience and prior familiarity with the SATD and testing literature.

As shown in Table 5, our instance-level analysis reveals a trend distinct from the static snapshots in RQ1. Initially, we identified 649 instances where SATD removal and test addition co-occurred within the same commit. However, our rigorous manual inspection revealed that some of these were not genuine SATD removals, such as file renamings, misidentified comments, or instances where the SATD was unresolved.

After excluding these cases, we confirmed 517 valid co-occurring instances. Among all 5,175 SATD removals, only 174 instances (3.4%) were verified as having new tests specifically targeting the removed SATD logic. The gap between the overall co-addition rate (12.5%)

■ **Table 6** Statistics of Bug-Inducing Commit Identification via LLM4SZZ.

Project	Bug-Fixing Commits	With Bug- Inducing	Rate
COMMONS LANG	424	263	62.0%
COMMONS IO	232	108	46.6%
HIBERNATE	3,199	2,300	71.9%
DUBBO	1,503	1,074	71.5%
SPOON	1,152	977	84.8%
MAVEN	1,030	820	79.6%
STORM	717	590	82.3%
JFREECHART	74	46	62.2%
Total	8,331	6,178	74.2%

and the verified rate (3.4%) indicates that even when tests appear in the same commit as an SATD removal, most are not directed at the resolved logic. Such co-occurring tests are typically peripheral changes (touching unrelated test fixtures, fixing pre-existing test failures, or covering other features modified in the same commit) rather than targeted verification of the debt resolution.

This gap varies considerably across projects. SPOON and DUBBO show the largest drops, with SPOON falling from 50.7% co-added to 19.4% verified and DUBBO from 39.3% to 8.0%. In these two projects, developers frequently bundle test changes with SATD removal commits, but most of those tests target unrelated logic. HIBERNATE, despite contributing the largest absolute number of verified instances (123), shows a low per-removal verified rate of 3.9%, suggesting that even projects with extensive testing infrastructure rarely add targeted verification at the moment of debt resolution.

Answer to RQ2

While SATD instances are generally covered by existing tests (RQ1), our manual verification reveals that only 3.4% of SATD removals are accompanied by new tests specifically tailored to the debt resolution. This suggests that SATD removal is predominantly performed without targeted verification, relying instead on a pre-existing safety net that may not be specifically optimized for the updated implementation.

***RQ₃*: What is the impact of removing SATD without adequate testing on software quality?**

To establish a foundation for identifying defects potentially introduced by SATD removal, we first characterized the bug dataset collected using LLM4SZZ. As shown in Table 6, we analyzed a total of 8,331 bug-fixing commits across eight open-source Java projects. From these, our pipeline successfully identified at least one bug-inducing commit for 6,178 bug-fixes, yielding an overall identification rate of 74.2%. This high success rate provides a sufficient corpus for tracing defects back to their origins and examining their proximity to SATD removal events.

The identification process utilized two complementary strategies: direct Large Language Model (LLM) analysis of patch content and a combination of traditional SZZ with LLM-based validation. The direct analysis (S1) succeeded in 58.5% of the files, while the SZZ+LLM

approach (S2) succeeded in 59.4%. Both strategies converged on the same result in 46.4% of the cases, demonstrating the consistency of our methodology. In total, we identified 10,933 unique bug-inducing commits, linked to 132,572 individual buggy statements.

Our analysis in RQ2 established that only 3.4% of all SATD removals (174 instances) are manually validated as being targeted by dedicated test modifications. To assess whether this scarcity influences subsequent defects, we compared bug induction rates between the test-added and no-test groups across multiple observation windows, excluding the 127 false positives filtered out during the RQ2 manual validation. Table 7 summarizes the results. Across short-to-medium-term intervals (30, 90, and 180 days), the bug induction rates are nearly identical between the two groups (1.72% vs. 1.89%, 2.30% vs. 2.36%, and 2.87% vs. 2.89%, respectively). A slight divergence becomes visible over longer windows, with the test-added group reaching 6.32% versus 4.37% when no time limit is imposed.

These results are counterintuitive: one might expect tests added during debt resolution to suppress subsequent defects. The bugs counted here, however, are those eventually identified and fixed in the same method as the SATD removal, not necessarily defects caused by the resolution itself, which complicates causal interpretation. Several factors may contribute to the observed pattern. The accompanying tests may not have been sufficient to verify the resolved logic. Developers may also add tests selectively to SATDs that are particularly critical or complex, whose inherent difficulty produces defects despite the presence of tests. Disentangling these mechanisms is beyond the scope of this study and warrants future investigation, possibly motivating the development of SATD-aware test prioritization techniques.

■ **Table 7** Bug-Inducing Commits Linked to SATD Removals across Observation Windows.

Metric	Tests Added	No Tests
Within 30 days	3 (1.72%)	92 (1.89%)
Within 90 days	4 (2.30%)	115 (2.36%)
Within 180 days	5 (2.87%)	141 (2.89%)
No time limit	11 (6.32%)	213 (4.37%)

Answer to RQ3

SATD resolutions accompanied by test modifications show no immediate reduction in linked bug induction rates within short-to-medium-term windows (1.72% vs. 1.89% within 30 days), and a slight long-term gap (6.32% vs. 4.37% with no time limit) is difficult to attribute to a single cause. The observed pattern may reflect insufficient test targeting, selection effects toward particularly complex SATDs, or tests surfacing latent defects that would otherwise go unrecorded.

5 Discussion and Implications

This section reflects on the three results presented above and considers what they imply for SATD management and software testing practice. We first interpret the patterns observed in test coverage, co-evolution, and defect alignment, and then derive implications for researchers and practitioners.

Interpretation of Results. The results suggest that the relationship between technical debt and testing is not straightforward. While one might expect developers to be more careful with code they know is suboptimal, our analysis in RQ1 indicates that this only happens in a minority of cases. Most projects follow a pattern of General Postponement, where technical debt is tested no more (and sometimes less) than the rest of the codebase. This suggests that the same factors leading to the introduction of technical debt, such as time pressure, likely also lead to insufficient testing of that code.

A particularly interesting observation is the Targeted Postponement pattern, exemplified by the Commons Lang project. In this case, the project maintains very high branch coverage, yet the specific execution paths containing technical debt are completely untested. This happens because tests are focused on “failure paths” (e.g., verifying that exceptions are thrown) to satisfy coverage metrics, while the “success paths” where the technical debt resides are ignored. This finding suggests that high-level coverage metrics can provide a false sense of security, as they do not necessarily reflect the testing quality of the most risk-prone parts of the system.

The longitudinal analysis in RQ2 and RQ3 reveals a disconnect in how testing accompanies the SATD lifecycle. Developers rarely synchronize test modifications with debt resolution, and the long-term differences in linked bug rates between the test-added and no-test groups admit multiple non-exclusive interpretations: tests may be insufficient to verify the resolved logic, developers may add tests selectively to particularly complex SATDs, or tests may surface latent defects that would otherwise go unrecorded. While we cannot adjudicate among these interpretations within the present data, they converge on a common implication: SATD repayment is currently treated as an ordinary code change rather than a high-risk maintenance activity warranting dedicated verification.

Implications. Our results converge on the implication that SATD repayment requires more deliberate verification than it currently receives. This finding has several implications for both researchers and practitioners.

For researchers, our study suggests several directions for future work. Since tests accompanying SATD removal may be insufficient to exercise the resolved logic, researchers should develop SATD-aware test generation techniques that produce tests targeting the specific code paths involved in the debt. Because developers also appear to add tests selectively, often to particularly complex SATDs, SATD prioritization techniques that identify which debt items most warrant verification effort would help focus testing resources. Furthermore, the impact of CI/CD coverage thresholds on test quality warrants further investigation: our findings suggest that high global coverage does not necessarily translate to a robust safety net, and strict thresholds may incentivize developers to create tests merely to satisfy numerical requirements rather than to verify complex debt-ridden areas.

For practitioners, these findings offer practical guidance for debt management. Project managers should not rely solely on project-wide coverage percentages, which can mask the lack of validation in critical, debt-heavy areas. Instead, they should supplement these with local coverage analysis of code marked with `TODO` or `FIXME` tags. Furthermore, organizations should treat SATD resolution as a high-risk maintenance activity warranting dedicated verification, rather than relying on existing safety nets that may not be adequate for the resolved logic.

6 Related Work

6.1 Empirical Studies of Testing

Large-scale empirical studies have consistently found a weak positive correlation between test volume and defect counts across open-source projects. *Kochhar et al.* [21] examined over 20,000 open-source projects, finding that projects with more code and more developers tend to have more test cases, and identifying a weak positive correlation between test volume and defect counts. *Islam et al.* [18] revisited these findings in a decade-long longitudinal study of Java projects, confirming that while testing grows alongside the codebase, this weak correlation remained stable through 2021. Beyond static metrics, *Beller et al.* [8] used IDE instrumentation to show that testing is frequently treated as an afterthought rather than a proactive activity.

The timing of test changes relative to production code has also been studied: while *Marsavina et al.* [25] established fundamental co-evolution patterns, *Wang et al.* [35] found that co-evolutions predominantly occur in the same commit (median $\approx 47\%$) and that their frequency decreases over time. Comparative studies on TDD vs. test-last development [13, 34] further show that different methodologies yield distinct coverage profiles: TDD tends to encourage higher branch coverage through refined test cases, while test-last development results in broader method-level coverage. Despite these insights, existing literature rarely addresses whether these practices apply specifically to code explicitly marked as technical debt, which is the gap our work fills.

6.2 Self-Admitted Technical Debt

Since *Potdar and Shihab* [29] formally introduced SATD, showing it appears in 2.4%–31% of source files with 26–64% eventually removed, substantial work has accumulated on identification and classification. Task tags (TODO, FIXME, XXX) have been established as reliable SATD signals [14]; complementary efforts include ontologies standardizing debt vocabulary [2], ML-based methods for distinguishing design from requirement debt [11], and LLM-based approaches for fine-grained classification [32]. Regarding lifecycle and impact, *Wehaibi et al.* [36] showed that while SATD-related changes do not always induce more immediate defects, they are more complex and difficult to perform, thereby hindering future modifications; *Mensah et al.* [26] further identified inadequate testing (21.2%) as one of the leading drivers of SATD, directly motivating our study.

Maldonado et al. [10] found that the majority of SATD is eventually resolved, frequently by the same developer who introduced it, persisting between 82–613 days on average before removal. To manage the resulting volume, *Lima et al.* [12] proposed a hybrid prioritization approach combining SATD descriptions with static analysis findings, achieving greater precision than comment-text-only methods. Despite these advances, while *Mensah et al.* identified inadequate testing as a *cause* of SATD, there remains a lack of empirical evidence on how testing practices are employed to safeguard the repayment process, which is the focus of this research.

7 Threats to Validity

Construct validity. concerns whether our metrics and procedures accurately capture the phenomena we intended to study. One potential threat to construct validity involves our use of tracer-line instrumentation as a proxy for measuring the test coverage of SATD instances.

While inserting an executable statement directly into the debt-ridden code block provides a precise way to verify execution, it does not guarantee that the tests fully exercised the specific logic described in the technical debt comment. For example, a tracer line confirms that a code block was reached, but not necessarily that all execution sub-paths within that block were covered. We mitigated this by manually mapping tracers to the most relevant lines within the debt-ridden block when automatic insertion was not possible.

Another threat related to construct validity is our method for linking SATD removal to subsequent defects. Our primary analysis imposes no time limit, treating any bug introduced in the same method as potentially related to the SATD removal. We report results at 30, 90, and 180-day windows as sensitivity analyses. While the no-limit approach is most conservative about excluding relevant bugs, it may also include bugs introduced by unrelated changes to the same method. Additionally, we cannot fully rule out that the higher bug-detection rate in the test-added group is partially explained by greater change complexity; a controlled analysis matching commits by churn size would be needed to isolate the effect of test addition from the effect of commit scale.

Internal validity. refers to the extent to which the observed results are due to the factors under study rather than confounding variables. The accuracy of the automated tools used in our study, such as DebtHunter, SATDBailiff, and LLM4SZZ, represents a potential threat to internal validity. To mitigate errors in SATD detection, we manually validated the results on a subset of our projects and excluded identified false positives. For the identification of bug-inducing commits, we used LLM4SZZ, which employs large language models that can exhibit stochastic behavior. We addressed this by using its dual-strategy approach and confirming the consistency between independent identification strategies.

There is also a risk of human error during the manual mapping of tracer lines for complex cases, such as inappropriate comment locations or syntax errors. To minimize this risk, we followed a protocol where instrumentation was carefully reviewed to ensure that the tracer line accurately reflected the execution of the logic referred to by the SATD comment, rather than just the literal position of the original comment.

External validity. concerns the generalizability of our findings to other contexts. We analyzed eight open-source Java projects from different domains and of varying scales. However, our findings may not generalize to commercial software, projects written in other programming languages, or systems with different testing cultures. We tried to broaden the scope of our observations by including projects with both pre-existing and manually integrated testing infrastructures.

The scope of the SATD instances we analyzed also presents a threat to generalizability. Our study was strictly limited to technical debt found within method bodies, which was necessary to measure executable coverage precisely. This means we excluded technical debt documented at the class or package level, such as architectural debt. Consequently, our conclusions primarily apply to technical debt related to implementation and local design rather than high-level system architecture.

To ensure the reliability and replicability of our study, we relied on established open-source tools including Jacoco, DebtHunter, and SATDBailiff. We have also provided a comprehensive replication package that contains our datasets, instrumentation scripts, and analysis results to allow other researchers to verify and build upon our work.

8 Conclusion

We empirically investigated the relationship between SATDs and software testing in eight open-source Java projects, examining 784 SATD instances for cross-sectional test coverage and 5,175 SATD removal events for longitudinal co-evolution and defect alignment.

Our central finding is a co-evolution gap between debt resolution and test maintenance. Although 60.7% of SATD-affected code is covered by existing tests, only 3.4% of SATD removals are accompanied by tests specifically targeting the resolved debt (vs. 12.5% that co-add any tests). Developers, therefore, treat SATD repayment as an ordinary code change rather than as a high-risk maintenance activity, despite their own prior flag.

Moreover, these rare test additions show no evidence of preventing short- to medium-term regressions: bug induction rates within 30, 90, and 180 days are nearly identical across the test-added and no-test groups (*e.g.*, 1.72% vs. 1.89% within 30 days). This null effect need not mean tests are ineffective in general; developers may selectively add tests to particularly difficult SATDs, whose complexity dominates the outcome. Separating test effectiveness from the difficulty of the underlying debt, and designing SATD-aware verification that prioritizes the hardest cases, are promising future directions.

Data Availability Statement

To support the reproducibility of our study, we have made our collection pipelines publicly available as replication packages. Throughout this research, we comprehensively utilized two repositories to build our foundational pipeline: the customized version of DebtHunter used for method-level SATD detection and precise metadata extraction [37], and the history mining framework that integrates our customized DebtHunter engine into SATDBailiff to trace SATD-removing events throughout the project lifecycle [38]. For the long-term defect analysis in RQ3, the replication package containing our execution scripts and environment for defect tracking and LLM4SZZ [33] analysis is provided at [16].

References

- 1 Eman Abdullah AlOmar, Ben Christians, Mihal Busho, Ahmed Hamad AlKhalid, Ali Ouni, Christian D. Newman, and Mohamed Wiem Mkaouer. SATDBailiff-mining and tracking self-admitted technical debt. *Science of Computer Programming*, 213:102693, 2022. doi: 10.1016/J.SCIC0.2021.102693.
- 2 Nicolli S. R. Alves, Leilane Ferreira Ribeiro, Viviane Caires, Thiago Souto Mendes, and Rodrigo O. Spínola. Towards an ontology of terms on technical debt. In *Proceedings of the Sixth International Workshop on Managing Technical Debt, MTD@ICSME 2014*, pages 1–7, 2014. doi:10.1109/MTD.2014.9.
- 3 Apache Software Foundation. Apache commons io. <https://github.com/apache/commons-io>. Accessed: 2026-05-17.
- 4 Apache Software Foundation. Apache commons lang. <https://github.com/apache/commons-lang>. Accessed: 2026-05-17.
- 5 Apache Software Foundation. Apache dubbo. <https://github.com/apache/dubbo>. Accessed: 2026-05-17.
- 6 Apache Software Foundation. Apache maven. <https://github.com/apache/maven>. Accessed: 2026-05-17.
- 7 Apache Software Foundation. Apache storm. <https://github.com/apache/storm>. Accessed: 2026-05-17.

- 8 Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer testing in the IDE: patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering*, 45(3):261–284, 2019. doi:10.1109/TSE.2017.2776152.
- 9 Aaditya Bhatia, Foutse Khomh, Bram Adams, and Ahmed E. Hassan. An empirical study of self-admitted technical debt in machine learning software. *CoRR*, abs/2311.12019, 2023. doi:10.48550/arXiv.2311.12019.
- 10 Everton da S. Maldonado, Rabe Abdalkareem, Emad Shihab, and Alexander Serebrenik. An empirical study on the removal of self-admitted technical debt. In *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution*, pages 238–248, 2017. doi:10.1109/ICSME.2017.8.
- 11 Everton da S. Maldonado, Emad Shihab, and Nikolaos Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062, 2017. doi:10.1109/TSE.2017.2654244.
- 12 Bruno Santos de Lima, Rogério Eduardo Garcia, and Danilo Medeiros Eler. Toward prioritization of self-admitted technical debt: an approach to support decision to payment. *Software Quality Journal*, 30(3):729–755, 2022. doi:10.1007/S11219-021-09578-7.
- 13 Davide Fucci, Hakan Erdogmus, Burak Turhan, Markku Oivo, and Natalia Juristo. A dissection of the test-driven development process: Does it really matter to test-first or to test-last? *IEEE Transactions on Software Engineering*, 43(7):597–614, 2017. doi:10.1109/TSE.2016.2616877.
- 14 Zhaoqiang Guo, Shiran Liu, Jinping Liu, Yanhui Li, Lin Chen, Hongmin Lu, and Yuming Zhou. How far have we progressed in identifying self-admitted technical debts? A comprehensive empirical study. *ACM Transactions on Software Engineering and Methodology*, 30(4):45:1–45:56, 2021. doi:10.1145/3447247.
- 15 Hibernate Community. Hibernate orm. <https://github.com/hibernate/hibernate-orm>. Accessed: 2026-05-17.
- 16 Kosei Horikawa. Bug induction and LLM4SZZ analysis scripts for SATD. <https://github.com/Mont9165/bug-analysis-for-satd>, 2026. Accessed: 2026-07-18.
- 17 Inria. Spoon. <https://github.com/INRIA/spoon>. Accessed: 2026-05-17.
- 18 Anisha Islam, Nipuni Tharushika Hewage, Abdul Ali Bangash, and Abram Hindle. Evolution of the practice of software testing in java projects. In *Proceedings of the 20th IEEE/ACM International Conference on Mining Software Repositories*, pages 367–371. IEEE, 2023. doi:10.1109/MSR59073.2023.00057.
- 19 Marko Ivankovic, Goran Petrovic, René Just, and Gordon Fraser. Code coverage at Google. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 955–963, 2019. doi:10.1145/3338906.3340459.
- 20 JFree Project. Jfreechart. <https://github.com/jfree/jfreechart>. Accessed: 2026-05-17.
- 21 Pavneet Singh Kochhar, Tegawendé F. Bissyandé, David Lo, and Lingxiao Jiang. An empirical study of adoption of software testing in open source projects. In *Proceedings of the 13th International Conference on Quality Software*, pages 103–112, 2013. doi:10.1109/QSIC.2013.57.
- 22 J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.
- 23 Stanislav Levin and Amiram Yehudai. The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes. In *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution*, pages 35–46. IEEE Computer Society, 2017. doi:10.1109/ICSME.2017.9.
- 24 Erin Lim, Nitin Taksande, and Carolyn B. Seaman. A balancing act: What software practitioners have to say about technical debt. *IEEE Software*, 29(6):22–27, 2012. doi:10.1109/MS.2012.130.
- 25 Cosmin Marsavina, Daniele Romano, and Andy Zaidman. Studying fine-grained co-evolution patterns of production and test code. In *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 195–204, 2014. doi:10.1109/SCAM.2014.28.

- 26 Solomon Mensah, Jacky Keung, Jeffrey Svajlenko, Kwabena Ebo Bennin, and Qing Mi. On the value of a prioritization scheme for resolving self-admitted technical debt. *Journal of Systems and Software*, 135:37–54, 2018. doi:10.1016/J.JSS.2017.09.026.
- 27 Ibuki Nakamura, Yutaro Kashiwa, Bin Lin, and Hajimu Iida. Understanding self-admitted technical debt in test code: An empirical study. *ACM Transactions on Software Engineering and Methodology*, 2026.
- 28 Francis Palma, Tamer Abdou, Ayse Bener, John Maidens, and Stella Liu. An improvement to test case failure prediction in the context of test case prioritization. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE 2018*, pages 80–89. ACM, 2018. doi:10.1145/3273934.3273944.
- 29 Aniket Potdar and Emad Shihab. An exploratory study on self-admitted technical debt. In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*, pages 91–100, 2014. doi:10.1109/ICSME.2014.31.
- 30 Gilberto Recupito, Vincenzo De Martino, Dario Di Nucci, and Fabio Palomba. A first look at the lifecycle of DL-specific self-admitted technical debt. In *Proceedings of the 2025 IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 150–157, 2025. doi:10.1109/SANER-C66551.2025.00030.
- 31 Irene Sala, Antonela Tommasel, and Francesca Arcelli Fontana. 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)*, pages 278–283. ACM, 2021. doi:10.1145/3463274.3464455.
- 32 Mohammad Sadegh Sheikhaei, Yuan Tian, Shaowei Wang, and Bowen Xu. An empirical study on the effectiveness of large language models for SATD identification and classification. *Empirical Software Engineering*, 29(6):159, 2024. doi:10.1007/S10664-024-10548-3.
- 33 Lingxiao Tang, Jiakun Liu, Zhongxin Liu, Xiaohu Yang, and Lingfeng Bao. LLM4SZZ: Enhancing SZZ algorithm with context-enhanced assessment on large language models. In *Proceedings of the 34th International Symposium on Software Testing and Analysis (ISSTA '25)*, pages 343–365. ACM, 2025. doi:10.1145/3728885.
- 34 Ayse Tosun, Muzamil Ahmed, Burak Turhan, and Natalia Juristo. On the effectiveness of unit tests in test-driven development. In *Proceedings of the 2018 International Conference on Software and System Process*, pages 113–122, 2018. doi:10.1145/3202710.3203153.
- 35 Sinan Wang, Ming Wen, Yepang Liu, Ying Wang, and Rongxin Wu. Understanding and facilitating the co-evolution of production and test code. In *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 272–283. IEEE, 2021. doi:10.1109/SANER50967.2021.00033.
- 36 Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. 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*, pages 179–188, 2016. doi:10.1109/SANER.2016.72.
- 37 Suzuka Yoshimoto. Modified DebtHunter tool for method-level SATD detection. <https://github.com/suu-y/DebtHunter-Tool>, 2026. Accessed: 2026-07-18.
- 38 Suzuka Yoshimoto. SATDBailiff framework integrating customized DebtHunter engine for history mining. <https://github.com/suu-y/SATDBailiff-DebtHunter>, 2026. Accessed: 2026-07-18.
- 39 Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011. doi:10.1007/S10664-010-9143-7.