

Improving Our Understanding of Test Code:
Technical Debt and Evolution

by

Md Shahidul Islam

A Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
of the University of Manitoba
in partial fulfilment of the requirements of the degree of

Master of Science

Department of Computer Science
University of Manitoba
Winnipeg

Copyright © 2026 by Md Shahidul Islam

Thesis advisor
Shaiful Chowdhury

Author
Md Shahidul Islam

Improving Our Understanding of Test Code: Technical Debt and Evolution

Abstract

Test code is essential for software quality, yet software maintenance research has focused mainly on production code, leaving important aspects of test-code quality and evolution underexplored. This thesis examines two dimensions of test-code maintenance: self-admitted technical debt (SATD) and method-level evolution. To understand SATD in test code, comments from open-source Java projects were manually analyzed and classified into a taxonomy that includes both source-code-like and test-specific categories. The analysis shows that test-code SATD resembles source-code SATD, while also capturing test-specific concerns absent from existing taxonomies. Evaluation of existing SATD detection tools and both open-source and proprietary large language models showed that these approaches are unreliable for detecting SATD in test code. To understand method-level test-code evolution, reliable method histories are needed. However, existing method-history generation tools have been evaluated mainly on production methods and against inaccurate oracles, leaving their reliability for test methods uncertain. To address this, updated and new oracles are constructed, and HistoryFinder is introduced as a method-history generation tool that achieves the highest overall accuracy with competitive runtime and outperforms other research-based tools. To examine test-production relationships, a test-to-production method-linking ground truth was constructed and used to evaluate existing linking techniques. Using these links and method histories, the analysis shows that some test methods are revised more frequently than their corresponding production methods and that several test-smell types are more prevalent among highly revision-prone test methods. Overall, this work contributes ground truths, tools, and empirical evidence for understanding test-code quality and evolution.

Preface

This thesis is based on three research manuscripts: one accepted for publication, one under revision, and one under review. I was the main contributor to these manuscripts under the supervision of Professor Shaiful Chowdhury. I designed the methodology in consultation with Professor Shaiful Chowdhury, conducted the experiments, analyzed the results, and contributed to the writing of the manuscripts. In some cases, my collaborators assisted with the annotation of the data.

Chapter 2 of this thesis is based on the following manuscript:

- S. Islam, M. N. I. Opu, S. Wang, and S. Chowdhury, “A First Look at Self-Admitted Technical Debt in Test Code: Taxonomy and Detection,” manuscript under revision at The Journal of Systems & Software, 2026.

Chapter 3 of this thesis is based on the following manuscript:

- S. Islam, A. Aowal, M. S. Uddin, and S. Chowdhury, “HistoryFinder: Advancing Method-Level Source Code History Generation with Accurate Oracles and Enhanced Algorithm,” manuscript accepted for publication in the proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE), Rio de Janeiro, Brazil, 2026.

Chapter 4 of this thesis is based on the following manuscript:

- S. Islam, A. Waseem, M. N. I. Opu, and S. Chowdhury, “Understanding Method-Level Test Code Evolution,” manuscript under review for the pro-

ceedings of the 49th IEEE/ACM International Conference on Software Engineering (ICSE), Dublin, Ireland, 2027.

Below is a list of my other co-authored publications completed during my MSc degree but not included in this thesis.

- M. N. I. Opu, S. Islam, M. Asaduzzaman, and S. Chowdhury, “How Do Agentic AI Systems Address Performance Optimizations? A BERTopic-Based Analysis of Pull Requests,” manuscript accepted for publication in the proceedings of the 23rd International Conference on Mining Software Repositories (MSR), Mining Challenge Track, Rio de Janeiro, Brazil, 2026.
- M. N. I. Opu, S. Islam, S. Rouhani, and S. Chowdhury, “Understanding the Issue Types in Open Source Blockchain-Based Software Projects with the Transformer-Based BERTopic,” manuscript accepted for publication in the proceedings of the 2025 IEEE International Conference on Collaborative Advances in Software and Computing (CASCON), 2025.

Acknowledgements

First and foremost, I express my deepest gratitude to Almighty God for giving me strength, patience, and perseverance in all aspects of my life.

I am grateful to my supervisor, Dr. Shaiful Chowdhury, for his guidance, encouragement, valuable feedback, and continuous support throughout this research journey. His advice and mentorship have played an important role in shaping this work.

I would also like to thank my thesis committee members, Dr. Shaowei Wang and Dr. Hamid Mansoor, for their time and support.

I also gratefully acknowledge the people of Bangladesh, whose contributions through public funding supported my undergraduate education. That support helped build the foundation that made this academic journey possible.

I would like to thank my lab mates for their support, discussions, and encouragement throughout this journey.

I express my appreciation to my friends and industry colleagues, especially Minaoar Hos-sain Tanzil, whose encouragement and support helped shape the beginning of this academic journey. Their collaboration, encouragement, and professional experiences contributed to my growth and helped prepare me for this research journey.

I am deeply thankful to my family and relatives for their love, sacrifices, constant encouragement, and motivation during difficult times.

Finally, I am deeply grateful to my beloved wife for her love, patience, understanding, and unwavering support throughout this journey.

To my mother, who has become a star in heaven, and whose tireless
work from dawn until late at night kept our family going.

To my father, who encouraged me to pursue education from childhood
and provided unwavering support despite enduring financial hardship
and uncertainty.

This work is dedicated to them with my deepest love, respect, and
gratitude.

Contents

Abstract	ii
Preface	iii
Acknowledgements	v
Dedication	vi
List of Tables	xi
List of Figures	xiii
Chapter 1: Introduction	1
1.1 Background and Motivation	2
1.1.1 Technical Debt	2
1.1.2 Method Evolution	3
1.2 Thesis Contributions	5
1.2.1 Self-Admitted Technical Debt in Test Code	5
1.2.2 Accurate Method History Generation	6
1.2.3 Test Method Evolution	6
1.3 Thesis Organization	7

Part I: Self-Admitted Technical Debt in Test Code **9**

Chapter 2: Self-Admitted Technical Debt in Test Code: Taxonomy and De-

tection	10
2.1 Introduction	12
2.2 Related Work	14
2.3 Methodology	18
2.3.1 Project Selection	19
2.3.2 Comment Parsing	20
2.3.3 Dataset Creation	21
2.3.4 Evaluation Metrics	23
2.4 Approach and Results	23
2.4.1 RQ1: What are the different types of SATD in test code?	24
2.4.2 RQ2: Can existing SATD detection tools detect test code SATD? . .	31
2.4.3 RQ3: Can open-source LLM detect SATD in test code?	36
2.4.4 RQ4: Can proprietary LLM detect SATD in test code?	40
2.5 Discussion	42
2.6 Threats to Validity	45
2.7 Conclusion	46

Part II: Method-History Generation and Test Method Evolution **47**

Chapter 3: HistoryFinder: Accurate Oracle Construction and Method-History

Generation	48
3.1 Introduction	50
3.2 Background and Related Work	53
3.2.1 Research on Code Change History and Maintenance	53

3.2.2	Research on History Construction	54
3.3	Developing the New Oracles	55
3.3.1	Updating the CodeShovel Oracle	57
3.3.2	Constructing the HistoryFinder Oracle	58
3.4	Developing the HistoryFinder Tool and Algorithm	58
3.4.1	Overview of HistoryFinder	61
3.4.2	The HistoryFinder Algorithm	62
3.5	Analysis and Results	67
3.5.1	RQ1: Difference with the corrected CodeShovel oracle	67
3.5.2	RQ2: Accuracy of the change history generation tools	69
3.5.3	RQ3: Runtime performance of the change history generation tools . .	72
3.6	Discussion	75
3.6.1	Threats to Validity	76
3.7	Conclusion and Future Work	77

Chapter 4: Test Method Evolution: Histories, Production-to-Test Mapping, and Test Smells

		79
4.1	Introduction	81
4.2	Related Work and Motivation	83
4.2.1	Software Evolution	84
4.2.2	Method-level history generation	85
4.2.3	Method-level production-to-test mapping	86
4.3	Experimental Setup	88
4.3.1	Project selection	88
4.3.2	Test and production method detection	88
4.4	Approach and Results	89
4.4.1	RQ1: Evaluation of existing history generation tools	89
4.4.2	RQ2: Production-to-test linking	93

4.4.3	RQ3: Revision in test and production methods	97
4.4.4	RQ4: Test smells and revisions	100
4.5	Discussion	104
4.6	Threats to Validity	106
4.7	Conclusion	106
Chapter 5: Conclusion and Future Work		108
5.1	Summary of Contributions	108
5.1.1	Characterizing Self-Admitted Technical Debt in Test Code	109
5.1.2	Generating Reliable Method-History	109
5.1.3	Understanding Test Method Evolution	109
5.2	Future Work	110
5.2.1	Improving Test-Code SATD Detection	110
5.2.2	Extending Method-History Generation	111
5.2.3	Building History-Aware Test Maintenance Tools	111
5.3	Concluding Remarks	111
Bibliography		113
Contributions of Authors		140
Manuscript 1 (Chapter 2): A First Look at the Self-Admitted Technical Debt in Test Code: Taxonomy and Detection		140
Manuscript 2 (Chapter 3): HistoryFinder: Advancing Method-Level Source Code History Generation with Accurate Oracles and Enhanced Algorithm		141
Manuscript 3 (Chapter 4): Understanding Method-Level Test Code Evolution		141

List of Tables

2.1	Training and testing datasets. Percentages in parentheses indicate the proportion within the corresponding datasets. Please note that, to enable fair comparisons across all tools and approaches, all accuracy metrics in all RQs were reported using only the test dataset.	23
2.2	SATD detection performance with/without retraining. The unsupervised MAT approach achieved the highest F1-score. Pattern-based methods had high precision but very low recall, and the Text Mining (TM) Method showed low precision but relatively higher recall. We also show the minimum, maximum, and average of all the scores across all the tools and approaches. . . .	33
2.3	Precision (P), Recall (R), F1-score (F1), and Matthews Correlation Coefficient (MCC) for detecting SATD comments using Flan-T5 series open-source LLMs across 0-, 2-, and 4-shot settings.	39
2.4	Precision (P), Recall (R), F1-score (F1) and Matthews Correlation Coefficient (MCC) for detecting SATD comments using GPT and Gemini across 0-, 2-, and 4-shot settings.	41
3.1	Overview of the 20 Java repositories used for the <i>HistoryFinder</i> oracle, including commit counts, source files, methods, and popularity metrics (stars and forks).	59

3.2	Comparison between the corrected and original oracles. As the corrected oracle includes changes related to annotations, JavaDocs, and formatting, the differences are shown both including (<i>Incl.</i>) and excluding (<i>Excl.</i>) those changes.	68
3.3	Precision, recall, and F1 score for method history tracking tools across three oracles. TP (True Positives) indicates correctly identified change commits; FP (False Positives) refers to incorrectly included commits; FN (False Negatives) denotes missed commits that should have been identified.	71
3.4	Execution times (in seconds) for the five tools. <i>IntelliJ</i> is excluded because of the need for manual execution.	74
4.1	Accuracy of the method-history tracking tools. TP denotes correctly identified change commits, FP denotes incorrectly included commits, and FN denotes missed change commits.	91
4.2	Runtime of the history generation tools in seconds.	92
4.3	Production-to-test link prediction performance by dataset for all approaches.	95
4.4	The top part of the table shows results for comparison between the <i>NTR</i> and <i>MTR</i> methods. For example, NTR % reports the percent of methods in the <i>NTR</i> group with a particular smell. The bottom part shows the comparisons between the <i>NTR</i> and <i>HTR</i> methods for a few smells. The full table for the bottom part is available (/data/rqs/rq4/test-smell-difference.pdf). . .	102

List of Figures

2.1	Workflow for collecting, sampling, analyzing, classifying test code comments (RQ1), and evaluating automated SATD detection using existing tools (RQ2), open-source LLMs (RQ3), and proprietary LLMs (RQ4).	18
2.2	Distribution of stars, forks, watchers, commits, and the number of extracted test code comments across the selected projects.	20
2.3	Manual categorization of the 615 self-admitted technical debt instances in test code. The categories shown within the dotted red boundary contain SATDs that did not align with any of the existing categories found in the source-code SATD taxonomies compared against. Because some SATD comments were assigned to more than one type, the count shown in a parent node is not necessarily the sum of the counts shown in its descendant nodes.	25
2.4	LLM prompt template for detecting SATD comment.	37
3.1	The process used to generate accurate oracles. The union of the change commits reported by all tools was first created, missing commits were added, and false positives were then removed.	57
3.2	Comparison of execution times across three oracles for five tools. To enhance readability, the y-axis is limited to 20 seconds for the first two subplots (<i>CodeShovel</i> oracles) and 50 seconds for the third subplot (<i>HistoryFinder</i> oracle).	73

4.1	Runtime of the studied tools. The y-axis is capped at 12s for readability; triangles indicate values above the limit, with numbers showing their counts.	92
4.2	Figure (a) shows the CDF of the number of revisions for both production and test methods. To enhance readability, the x-axis is restricted to 50. Figure (b) shows the CDF of correlation coefficients between the number of revisions in test methods and their corresponding production methods. Figure (c) shows the CDF of differences in revisions between test methods and their corresponding production methods. It is important to note that all test and production methods were used for Figure (a), while only mapped examples were used for (b) and (c).	98
4.3	Size-based analysis. Figure (a) shows the difference in percent with 95% confidence interval, and Figure (b) shows the odds ratio (OR). The comparison is for <i>MTR</i> and <i>NTR</i> methods, as well as <i>HTR</i> and <i>NTR</i> methods. The selected 11 types of smells are joined together and considered as one smell.	103

Chapter 1

Introduction

Software maintenance is a continuous and expensive part of the software life cycle. After initial development, software systems must be corrected, adapted, improved, and extended to keep them operational, reliable, and aligned with changing requirements, platforms, dependencies, and user expectations [1–4]. This maintenance effort applies not only to production code but also to test code. Since test code is used to check the correctness of production code, its own correctness and maintainability are equally important. If tests are incorrect, outdated, or difficult to maintain, they may fail to detect regressions, report false failures, or provide developers with misleading confidence. Therefore, test code should not be treated as a secondary artifact; it deserves research and maintenance attention comparable to production code.

To support software maintenance, researchers have developed indicators and tools that help identify artifacts requiring developer attention, such as source-code metrics, code smells, change histories, and self-admitted technical debt [5–11]. However, much of this research has focused mainly on production code. For example, although many studies have examined self-admitted technical debt in source-code comments, less is known about how developers admit debt in test code. Similarly, although method-level history research has studied how production methods evolve, the evolution of test methods remains less understood. At the

same time, prior studies have shown that test code has its own quality and maintenance concerns, including test smells and cloning [12–16]. Together, these observations motivate a closer examination of SATD in test code and the evolution of test methods.

This thesis addresses these gaps by studying test-code maintenance from two complementary perspectives. First, it explores self-admitted technical debt (SATD) in test code, where developers explicitly record maintenance concerns, shortcuts, workarounds, or future work in comments. Second, it investigates method-history generation and method-level test-code evolution, focusing on how individual methods can be tracked accurately across revisions and how test methods evolve relative to the production methods they exercise. Together, these two perspectives provide a more detailed understanding of test-code maintenance and help build stronger empirical foundations for future research on test-code quality and evolution.

1.1 Background and Motivation

1.1.1 Technical Debt

Technical debt refers to suboptimal implementation decisions that may provide short-term benefits but increase future maintenance costs [17]. A particular form of technical debt is self-admitted technical debt (SATD), where developers explicitly document debt, shortcuts, workarounds, or postponed work in code comments [10]. Potdar and Shihab [10] were among the first to systematically study SATD in open-source projects, and Maldonado and Shihab [11] later classified SATD comments into categories such as design, requirement, defect, test, and documentation debt. Subsequent studies showed that SATD can persist over time [18], negatively affect software evolvability [19], and be associated with larger, more complex, less readable, more change-prone, and more bug-prone methods [20].

Because SATD captures maintenance concerns explicitly written by developers, it has received considerable research attention. Prior studies have proposed techniques for its detection [11, 21–27], classification [11, 28–30], mitigation, and longitudinal tracking [31–36].

Researchers have also extended SATD studies to specialized domains, including machine learning [29, 30], Dockerfiles [37], blockchain projects [38], and Android applications [39]. However, the subject systems and datasets used in most SATD studies have been either exclusively production code or overwhelmingly dominated by production code. Although Counsell and Swift [40] examined SATD in both test and production classes, their focus was the timing of SATD introduction rather than the characterization or detection of SATD in test code.

Relying on findings from production-code SATD to understand test-code SATD may lead to incomplete or inaccurate conclusions. Test code differs from production code in several important ways. Prior studies have shown that test code can contain test-specific code smells [12, 13], exhibit a higher prevalence of cloning [15], and contain comments less frequently than production code [14]. Moreover, developers may devote less attention to the quality of test code than to production code, which can result in obsolete, difficult-to-maintain, or low-quality tests [14, 16, 41–44]. These differences point to the possibility that SATD in test code may have different forms, causes, and detection challenges than SATD in production code.

Therefore, SATD in test code remains insufficiently understood. It is still unclear whether SATD in test code follows the same patterns as SATD in production code, whether production-oriented SATD taxonomies are sufficient for test code, and whether existing SATD detectors or large language models can reliably identify SATD in test comments. This motivates Chapter 2, which characterizes SATD in test code and evaluates the effectiveness of existing tools and large language models for detecting it.

1.1.2 Method Evolution

Software evolution histories provide valuable evidence for understanding how software systems change over time. Practitioners use code histories for provenance analysis, traceability, code comprehension, onboarding, code review, and expert identification [45], while re-

searchers use them for bug prediction, change prediction, clone management, technical debt analysis, authorship attribution, and maintenance effort estimation [7–9, 20, 46–49]. Version control systems support file-level analysis by recording which files change across commits, but they do not directly capture the histories of individual methods. Method-level evolution must therefore be reconstructed from file-level revisions.

One objective of this thesis is to understand how individual test methods evolve and how their evolution relates to the production methods they exercise. Addressing this objective requires reliable test-method histories. However, prior studies of production–test co-evolution have often lacked validated test-method histories and have therefore relied on custom heuristics or extensive manual inspection [50–57]. Several research tools have been developed to reconstruct method histories from Git repositories. For example, *Historage* [58] and *Finer-Git* [59] restructure repositories to support method-level analysis, whereas *CodeShovel* [45] and *CodeTracker* [60] reconstruct method histories on demand. Despite the availability of these tools, two important limitations remain. First, existing evaluation oracles may contain inaccuracies. Second, because method-history tools have been evaluated against these potentially inaccurate oracles, their reported accuracy may also be unreliable. Therefore, before studying test-method evolution, Chapter 3 validates existing method-history oracles, constructs new validated histories, reevaluates method-history tools using the corrected oracles, and introduces *HistoryFinder* to improve method-history accuracy while maintaining competitive runtime.

Since these tools have been evaluated mainly on production-code methods, their effectiveness must be assessed separately using validated test-method histories before they are applied to studies of test-code evolution. In addition, examining how test methods evolve in relation to the production methods they exercise requires identifying the corresponding production methods and comparing the histories of the linked test and production methods. Existing production-to-test mapping techniques employ both static and dynamic approaches [61–64], but they have often been evaluated using limited ground truth from a small number of

projects and may involve substantial precision–recall trade-offs. These limitations motivate evaluating production-to-test mapping approaches before combining them with validated test-method histories to study production-test method evolution.

Taken together, these limitations reveal three main research gaps. First, existing SATD research provides limited evidence about how developers admit technical debt in test code. Second, method-level history generation still suffers from oracle and tooling limitations. Third, test-code evolution has not been sufficiently studied using validated test method oracles and method-level production-to-test links. Without these foundations, it is difficult to determine whether test methods mainly evolve in response to production code or due to their own maintenance concerns. This thesis focuses on open-source Java systems and uses method-level analysis as the primary granularity because methods are small enough to capture fine-grained maintenance behavior but large enough to represent meaningful units of developer work.

1.2 Thesis Contributions

This thesis makes three main contributions, presented in Chapters 2, 3, and 4. The first contribution corresponds to Chapter 2, which examines self-admitted technical debt in test code. The second and third contributions correspond to Chapters 3 and 4, which first establish accurate method-history generation support and then use that support to examine method-level test-code evolution.

1.2.1 Self-Admitted Technical Debt in Test Code

The first contribution is a large-scale empirical study of SATD in test code. Chapter 2 manually analyzes 50,000 test-code comments sampled from 1.6 million comments across 1,000 open-source Java projects. SATD comments in test code are identified, classified into a taxonomy that includes test-specific categories, and used to evaluate existing SATD

detection tools as well as open-source and proprietary large language models.

The results show that test-code SATD has characteristics that are not fully captured by production-code SATD taxonomies or detectors. Existing SATD approaches show low recall, while the evaluated large language models struggle to provide reliable precision-recall trade-offs for this task. This contribution provides a manually curated dataset, a test-code SATD taxonomy, and empirical evidence that can support future research on test-aware SATD detection and classification.

1.2.2 Accurate Method History Generation

The second contribution is a methodological foundation for accurate method-level evolution research. Chapter 3 constructs systematically developed method-history oracles using a semi-automated process that combines multiple state-of-the-art tools with expert manual validation. It also introduces *HistoryFinder*, a method-level history generation tool designed to improve accuracy while maintaining competitive runtime performance.

The evaluation shows that *HistoryFinder* provides a strong overall balance of precision, recall, F1 score, and runtime among the evaluated research tools. This contribution improves both the benchmark data and the tool support needed for method-level maintenance research on change-proneness, bug-proneness, technical debt, and test-code evolution.

1.2.3 Test Method Evolution

The third contribution is a method-level empirical study of test-code evolution. Chapter 4 first evaluates whether state-of-the-art method-history tools can accurately track test method histories using a manually constructed oracle. It then evaluates method-level production-to-test mapping approaches utilizing existing benchmarks and a newly constructed oracle. Using the most suitable history-tracking and mapping approaches, the revision frequencies of test methods and their corresponding production methods are compared, and the association between test smells and highly revised test methods is examined.

The results challenge the common assumption that test code primarily changes in response to production code. A substantial portion of test methods undergo as many or more revisions than their corresponding production methods, and highly revised small test methods show stronger associations with test smells. This contribution provides a more precise empirical basis for future research on test maintenance, regression testing, obsolete test detection, and test smell management.

Overall, this thesis makes the following contributions:

- It provides a manually curated dataset and taxonomy of SATD in test code and evaluates the ability of existing SATD tools and large language models to detect test-code SATD.
- It constructs reliable method-level history oracles and introduces *HistoryFinder*, a method-history generation tool with strong accuracy and competitive runtime.
- It evaluates method-history tools on test methods and assesses method-level production-to-test mapping techniques using existing benchmarks and a newly constructed benchmark.
- It presents a method-level empirical study showing how test methods evolve relative to the production methods they exercise and how test smells are associated with highly revised test methods.

1.3 Thesis Organization

The remainder of this thesis is organized into two parts after this introductory chapter.

Part I, Self-Admitted Technical Debt in Test Code, contains Chapter 2, which examines how developers admit technical debt in test code and how effectively such debt can be detected. The chapter describes the collection and manual analysis of test-code comments,

develops a taxonomy of test-code SATD, and evaluates existing SATD detectors and large language models for identifying SATD in test comments.

Part II, Method-History Generation and Test Method Evolution, contains Chapters 3 and 4. This part develops the methodological support needed for method-level evolution research and then applies that support to study test methods. Chapter 3 discusses limitations in existing method-history oracles, describes the construction of new oracles, introduces *HistoryFinder*, and evaluates method-history tools. Chapter 4 then presents the study of method-level test-code evolution by evaluating method-history tools on test methods, evaluating production-to-test method linking approaches, comparing the revision frequencies of test and production methods, and analyzing the association between test smells and highly revised test methods.

Chapter 5 concludes the thesis by summarizing the findings from both parts, discussing their implications for test-code maintenance research, and outlining directions for future work.

Part I

Self-Admitted Technical Debt in Test Code

Chapter 2

Self-Admitted Technical Debt in Test Code: Taxonomy and Detection

This chapter contributes to the first part of this thesis, which examines technical debt in test code. It focuses on self-admitted technical debt (SATD), a form of technical debt explicitly documented by developers in source-code comments. In the context of test code, these comments capture maintenance concerns such as incomplete tests, postponed improvements, limitations, workarounds, and known quality issues. This chapter investigates how SATD appears in test code, develops a taxonomy of test-code SATD, and evaluates whether existing SATD detection techniques can effectively identify such debt.

This chapter is based on the following manuscript:

- S. Islam, M. N. I. Opu, S. Wang, and S. Chowdhury, “A First Look at the Self-Admitted Technical Debt in Test Code: Taxonomy and Detection,” manuscript under revision at The Journal of Systems & Software, 2026.

Abstract

Self-admitted technical debt (SATD) refers to comments in which developers explicitly acknowledge code issues, workarounds, or suboptimal solutions. SATD is known to significantly increase software maintenance effort. While extensive research has examined SATD in source code, its presence and impact in test code have received no focused attention, leaving a significant gap in understanding how SATD manifests in testing contexts.

SATD in test code is investigated by manually analyzing 50,000 comments randomly sampled from 1.6 million comments across 1,000 open-source Java projects. From this sample, after manual analysis and filtering, 615 SATD comments were identified and classified into 12 distinct categories, building a taxonomy of test code SATD. To investigate whether test code SATD can be detected automatically, existing SATD detection tools, as well as both open-source and proprietary LLMs, are evaluated. Among the existing tools, MAT performed the best, albeit with moderate recall. Both open-source and proprietary LLMs exhibited poor detection accuracy, primarily due to low precision. These results indicate that neither existing approaches nor current LLMs can reliably detect SATD in test code.

Overall, these results provide a large-scale analysis of SATD in test code, a nuanced understanding of its types, and the limitations of current SATD detection methods. The findings lay the groundwork for future research on test code-specific SATD.

2.1 Introduction

Software maintenance keeps applications reliable and adaptable as they evolve. It involves continuous updates to fix bugs, improve performance, and meet user needs, often demanding more effort than initial development [1, 2]. As such, to reduce future maintenance burden, researchers have developed methods to identify and manage code components that are more maintenance prone [5, 7, 8, 45, 65–67]. Previous research on software maintenance, however, has predominantly focused on production code (i.e., source code), with significantly less attention given to test code [68, 69]. This oversight is significant, as the quality and maintainability of test code also play a crucial role in overall software quality and long-term maintainability [16, 43, 70].

Consider the case of Self-Admitted Technical Debt (SATD) [10, 17, 19, 71–75], where developers explicitly acknowledge the need for improvement or rework through comments in the source code. SATD has been widely recognized as a factor that negatively impacts software maintainability. For example, a recent study by Chowdhury *et al.* [20] found that source code methods containing SATD not only exhibit lower code quality, but are also more change-prone and bug-prone than their non-SATD counterparts. Not surprisingly, considerable research has focused on detecting [11, 21–27], classifying [11, 28–30], and mitigating SATD [31–36]. Unfortunately, the subject samples in these studies have been either exclusively source code or overwhelmingly dominated by it.

Relying on findings about SATD in source code to understand SATD in test code may lead to inaccurate conclusions. This is because test code often exhibits characteristics that differ significantly from those of production code. For instance, test code tends to contain test-specific code smells [12, 13], a higher prevalence of code cloning [15], and a lower frequency of comments [14]. Moreover, developers typically devote less attention to the quality of test code than to production code, which can result in a range of quality issues within test suites [14, 16, 41–44]. These differences highlight the need to study SATD in the test code independently. Motivated by this gap, the study focuses specifically on the detection and

classification of SATD in test code.

To study test code SATD, over 1.6 million test code comments from 1,000 open-source Java projects were used. From this corpus, 50,000 comments were randomly sampled for manual labeling. The performance of existing SATD detection tools on test code comments was evaluated, and the effectiveness of Large Language Models (LLMs) for automated SATD detection was investigated. In general, the contribution of this chapter is founded on four research questions.

RQ1: What are the different types of SATD in test code?

After manually analyzing 50,000 comments and applying a filtering approach, 615 SATD comments were identified and classified into 12 distinct categories. While some of these categories—such as code debt and design debt—are consistent with those previously identified in source code SATD [11], four categories are unique to test code. An example is the limited test category, which captures SATD specific to incomplete or low-value test validation.

RQ2: Can existing SATD detection tools detect test code SATD?

Test code comments exhibit patterns and intentions that differ from those found in source code. As a result, the performance of the existing SATD detection tools remains unknown. Seven existing approaches are evaluated: pattern-based approach by Potdar et al. [10], Natural Language Processing (NLP)-based approach by Maldonado et al. [25], Text Mining (TM)-based approaches by Huang et al. [22] and Liu et al. [21] (SATD Detector), Matches task Annotation Tags (MAT)-based approach by Guo et al. [76], a multi-task learning model based on BERT proposed by Gu et al. [77], and DebtHunter by Sala et al. [78]. The results show that the MAT-based approach exhibits the best performance (0.75 F1-score), but with moderate recall.

RQ3: Can open-source LLMs detect SATD in test code?

Since SATD comments are written in natural language, LLMs may be better suited to detect them, particularly when comments lack recurring SATD keywords. Encouraged by Sheikhaei et al. [24], the Flan-T5 series (Small to XXL) was evaluated under 0 to 4-

shot settings using five different prompts. Results show that open-source LLMs struggle to reliably detect SATD in test code. While the best F1-scores reached 0.68 under MAT keywords suggested prompt, the recall was only 0.52.

RQ4: Can proprietary LLMs detect SATD in test code?

As proprietary LLMs can outperform open-source models in many scenarios [79], two proprietary families—Gemini and GPT—were evaluated under few-shot settings of 0, 2, and 4 shots. Interestingly, in contrast to the open-source LLMs, the tested proprietary LLMs exhibited very high recall but very low precision.

In summary, a taxonomy of test code SATD categories is developed, revealing diverse maintenance issues. Additionally, the results show that traditional tools, although they do not exhibit very good performance on test code SATD detection, perform better than both open-source and proprietary LLMs. These findings highlight the need for test-aware SATD detection methods that can improve both precision and recall.

To facilitate replication and extension, the dataset developed here is shared publicly.¹ This dataset can work as a benchmark for evaluating the accuracy of future detection and classification models and tools.

The remainder of this chapter is organized as follows. Section 2.2 discusses related work on SATD. Section 2.3 describes the data collection and manual SATD labeling process. Section 2.4 presents manual SATD classification, detection approaches, and the results for all research questions. Section 2.5 discusses the implications of the findings, while Section 2.6 outlines threats to validity. Finally, Section 2.7 concludes the chapter and highlights future directions.

2.2 Related Work

Software quality and maintainability have been extensively studied by the research community, covering a wide range of topics such as code quality metrics [80–82], change-proneness [5,

¹<https://github.com/SQMLab/technical-debt>

65, 83], bug-proneness [7, 8, 84], code readability [85, 86], and code understandability [87], among others. Due to its relevance to software quality and maintainability, a different stream of research has focused on SATD detection, classification, and its long-term maintenance impact.

Potdar and Shihab [10] were the first to systematically study SATD in four large open-source projects, without differentiating between production and test code. They found SATD in 2.4–31.0% of project files and investigated the reasons for its introduction and its removal frequency. Bavota and Russo [18] replicated the study of Potdar and Shihab, but by analyzing a large number of projects (159). They found that SATD comments tend to increase over time and persist for long periods. Chowdhury et al. [20] conducted a large-scale method-level analysis of 774,051 methods across 49 open-source projects, revealing that methods with SATD are larger, more complex, less readable, and more prone to bugs and frequent changes. Similarly, Wehaibi et al. [19] analyzed five major open-source systems and found that technical debt negatively affects software evolvability, making future changes more difficult. Collectively, these studies underscore the negative impact of SATD on long-term software health, highlighting the importance of its classification and detection.

The detection of SATD has been explored extensively through both manual and automated approaches. Potdar and Shihab [10] identified 62 frequent keywords showing their potential in SATD detection. Various automated techniques have been proposed as well. Guo et al. [76] introduced the Matches task Annotation Tags (MAT) method, which achieves performance comparable to NLP-based approaches [25]. Huang et al. [22] applied text mining to extract features, performed feature selection, and trained a composite classifier for SATD detection. More recently, large language models (LLMs) [24, 77, 88] have been employed, outperforming traditional methods for detecting SATD in application code.

To better conceptualize technical debt, Alves et al. [89] proposed an ontology synthesizing definitions and indicators scattered across the literature, categorizing technical debt into 13 types (e.g., architecture, build, code, test). Building on this foundation, Maldonado

and Shihab [11] manually reviewed 33,093 heuristically filtered source code comments from five Java projects and identified 2,457 SATD comments, which they classified into five categories: design, requirement, defect, test, and documentation. Maldonado et al. [25] later extended this study to additional projects and applied natural language processing (NLP) techniques for classification. Their dataset has been used by subsequent studies on automated SATD detection and classification [24, 76, 77, 90–92]. Recent work has expanded SATD research into new domains beyond traditional software projects. OBrien et al. [29] categorized SATD comments in machine learning code into 23 distinct types, while Bhatia et al. [30] extended Bavota et al.’s [18] taxonomy by adding new machine-learning-specific categories and subcategories to classify SATD in machine learning software. Liu et al. [93] examined seven popular open-source deep learning frameworks to assess SATD prevalence and categories. Pinna et al. [38] found that SATD in blockchain projects is more prevalent than in non-blockchain systems. Azuma et al. [37] manually classified SATD comments in Dockerfiles, identifying five main classes—including Docker-specific categories—to capture domain-specific debt. Similarly, Wilder et al. [39] analyzed 15,614 open-source Android applications and found that code debt is the most prevalent type.

Although a study by Counsell and Swift [40] examined SATD in test code in addition to source code, their focus was on understanding whether the time of day is related to writing SATD, rather than on identifying patterns of test code SATD or supporting their detection. During the course of this chapter, a parallel publication by Nakamura et al. [94], which also investigated SATD in test code, became available. However, the analysis presented here differs substantially from theirs in several important aspects, as outlined below.

- To identify test-code SATD, 50,000 code comments are manually examined. In contrast, Nakamura et al. relied on an existing SATD detection tool, DebHunter, which was not specifically designed or trained to detect SATD in test code. The evaluation of DebHunter on test-code SATD detection revealed low accuracy (RQ2). As such, using DebHunter to construct ground-truth datasets, as done in [94], may have in-

troduced inaccuracies into the identified SATD instances and affected the reliability of the resulting analysis.

- While constructing their taxonomy of test-code SATD, Nakamura et al. did not attempt to align their findings with existing SATD taxonomies. Such comparisons are important because they help determine how newly identified categories differ from or extend established ones, and this practice is commonly followed in software engineering research. For example, in their work, “Why Attention Fails: A Taxonomy of Faults in Attention-Based Neural Networks”, Jahan et al. [95] introduced new categories only when the identified issues could not be adequately mapped to an existing deep-learning bug taxonomy. Similarly, each SATD instance was first mapped to the existing source-code SATD taxonomy, and new categories were introduced only when the comments could not be appropriately classified within the existing structure. Without such an analysis, it remains unclear whether a separate taxonomy for test-code SATD is necessary or whether existing SATD taxonomies are already sufficient to capture these instances.
- The problem of identifying whether a comment constitutes SATD should be addressed before attempting to classify it into specific SATD categories. If SATD comments cannot be reliably distinguished from non-SATD comments, then classifying their types becomes considerably less meaningful. However, Nakamura et al. focused on the multiclass classification task (i.e., predicting SATD categories) without first addressing the more fundamental binary classification task of distinguishing SATD from non-SATD comments. This concern is particularly significant because the results reported here show that existing SATD detection tools, as well as both open-source and proprietary LLMs, struggle to accurately identify SATD in test code. These findings suggest that reliable test code SATD detection remains an open challenge and should be established before pursuing prediction of SATD classes.

- In addition to evaluating a wide range of open-source and proprietary LLMs, state-of-the-art SATD detection tools are also evaluated. This comparison is motivated by the practical consideration that LLM-based solutions can be computationally expensive, and therefore should ideally be used only if simpler approaches do not provide sufficient performance.

Motivation: While numerous studies have examined different aspects of test code maintenance [16, 41–44, 96], investigating the detection and classification of SATD in test code remains underexplored. Motivated by this gap, an empirical analysis of test code SATD is conducted to explore its types and detection.

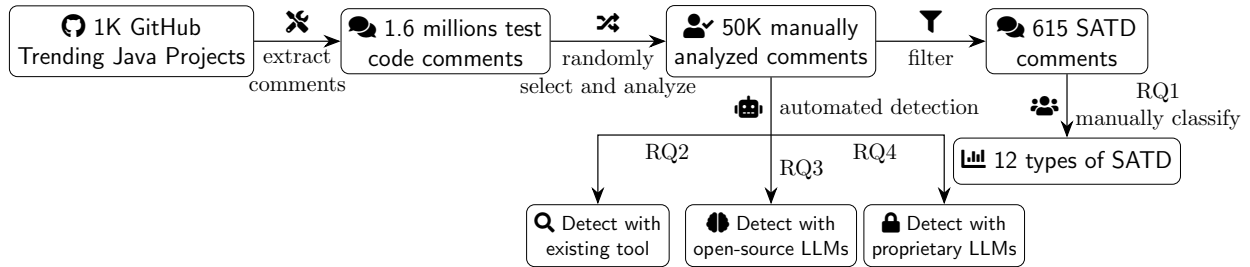


Figure 2.1: Workflow for collecting, sampling, analyzing, classifying test code comments (RQ1), and evaluating automated SATD detection using existing tools (RQ2), open-source LLMs (RQ3), and proprietary LLMs (RQ4).

2.3 Methodology

Figure 2.1 presents a high-level overview of the methodology, outlining the key stages involved in data collection, comment extraction, manual labeling, and SATD detection using both traditional and LLM-based approaches. It also clarifies how each branch maps to the research questions: manual taxonomy construction in RQ1 and automated detection pipelines in RQ2–RQ4.

2.3.1 Project Selection

The data collection aimed to gather comments from a diverse range of projects to avoid relying on a limited set of contributors, which could hinder the generalizability of the observations. Such generalizability is essential for developing a benchmark dataset, as selecting SATD from only a few projects would be strongly influenced by the commenting styles of a small number of individuals. This, in turn, would hinder the meaningful evaluation of current and future SATD detection tools. While selecting a large number of projects, however, it was important to ensure that the collected comments were not predominantly drawn from low-quality projects (e.g., toy projects) [97]. Therefore, GitHub’s trending project list was used, ranked by the number of stars retrieved via the GitHub API. Although Borges et al. [98] showed that star counts do not always correspond to sustained engagement or project quality, they also found that many practitioners continue to consider star counts before adopting a tool in their projects. Nevertheless, the potential negative impact of star-based project selection was mitigated by including another popularity indicator—projects with fewer than 100 forks were not included. In addition, 100 projects were randomly analyzed, and the selected repositories were found to be well-maintained.

At the end, 1,000 Java-based repositories were selected that were more likely to contain meaningful discussions and development practices relevant to SATD detection in test code. The focus was limited to a single programming language due to reliance on a parser for comment extraction—supporting multiple programming languages would require multiple parsers. Also, since the focus is on test code only, techniques were needed to exclude production code files (described in Section 2.3.2), which would have been more challenging if applied across multiple programming languages. The selected projects cover a wide range of software types, including libraries, frameworks, applications, and middleware. Figure 2.2 shows the distribution of stars, forks, watchers, commits, and extracted test code comment counts across these projects.

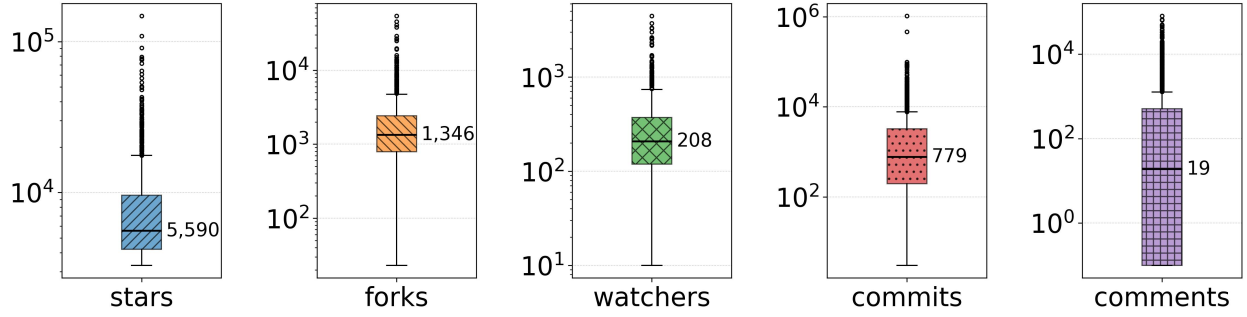


Figure 2.2: Distribution of stars, forks, watchers, commits, and the number of extracted test code comments across the selected projects.

2.3.2 Comment Parsing

The source code of each project was first retrieved from GitHub with an automated script. Since only test code is considered, it is necessary to distinguish comments in production code from those in test code. To identify test code, two heuristics were used. First, as is common in Java projects using Maven, Gradle, or Ant build systems, test code files and packages are typically placed under the `src/test/java` directory. Therefore, all Java files within this directory were treated as test code. Second, for files located outside the standard test directory, the presence of unit test annotations (e.g., `@Test`, `@Before`) from well-known unit testing frameworks such as JUnit, Mockito, and TestNG was checked.

For code parsing, `JavaParser`² was used to extract all associated comments—including block, line, and Javadoc types—along with their start and end line numbers. To preserve the continuity of comments, consecutive single-line comments were concatenated. This is important because developers often use multiple single-line comments to document a single concept, and failing to merge them would result in fragmented sentences and a loss of contextual meaning. As a result, the raw number of extracted comments does not directly correspond to the number of semantically distinct comments. After merging consecutive single-line comments, a total of 1,653,658 test code comments were obtained and stored in a relational database to enable efficient downstream processing. Among the 1,000 analyzed

²<https://github.com/javaparser/javaparser> last accessed: Oct 01, 2025

projects, 45 contained a substantial number of comments, each exceeding 10,000. An additional 140 projects had between 1,000 and 10,000 comments, while 497 projects had fewer than 1,000 comments. The remaining 318 projects contained no test code comments at all.

2.3.3 Dataset Creation

Since manually reviewing more than 1.6 million test comments to identify SATD comments will be extremely time-consuming, 50,000 comments were randomly selected. Similar to the original SATD study [10], the first author, a graduate student with 10+ years of industry experience as a lead software engineer, checked how many of these comments are SATD, making two classes of comments: SATD and non-SATD. This process required approximately 165 hours to complete. To assess labeling reliability, an independent double-check was conducted using a statistically grounded sample targeting a 95% confidence level. The first author provided the second author with 619 randomly selected comments, obtained through stratified sampling across the two classes (237 SATD and 382 non-SATD). The second author, another graduate student with 2+ years of industry experience as a software engineer, independently labeled these 619 samples. The Cohen’s kappa coefficient [99] was 0.95, implying an almost perfect [100] inter-rater agreement score and the reliability of the dataset labeling.

The manual review identified 1,024 SATD comments. However, among these 1,024 SATD comments, a striking 409 originated from a single project, OpenAPI Generator, accounting for approximately 38% of that project’s comments. Upon closer inspection, this repository was found to contain hundreds of autogenerated Java client samples (e.g., OkHttp, Retrofit) with templated SATD comments following the pattern *TODO*. Because this project disproportionately skewed the overall distribution of SATD across projects, including its comments would have hindered the generalizability of the benchmark dataset. For example, the taxonomy of SATD developed here (RQ1) would be affected, and any evaluation of SATD detection tools would be biased. Consequently, all 1,063 comments from this project were

excluded from the experimental dataset. Then, 943 non-English comments were removed. The refined dataset ultimately comprises 47,994 comments from 488 distinct projects, of which 615 (1.28%) are classified as SATD, originating from 146 projects.

The filtered dataset of 47,994 comments also contains 15,439 duplicate comments—unsurprising given practitioners often use similar keywords to indicate SATD [10]. Therefore, a second dataset was created by applying deduplication. This allows the model’s ability to make predictions on completely unseen comments to be evaluated more directly. Thus, the original dataset (the duplicate set) preserves the natural distribution of comments as found in the projects, while the latter dataset allows detection effectiveness to be evaluated without the risk of any data leakage. Clearly, and unsurprisingly, the dataset is highly imbalanced, with SATD instances comprising only 1.28% of the total data. As shown later, this severe imbalance makes accurate detection particularly challenging.

Some evaluations in RQ2 involve retraining the existing tools. As such, the dataset was divided into 80% training and 20% testing. To prevent data leakage, each project was kept exclusively in either the training or testing set, while also maintaining the proportional distribution of SATD and non-SATD comments across both sets—i.e., 20% of all SATD should be in the test, and the rest in the train dataset. To do so, the test set was constructed by randomly selecting one project at a time, measuring the distributions until approximately 20% of the total SATD and 20% of total non-SATD comments were included. The remaining comments formed the training set. Because of project-wise selection, the split between training and testing was not always exactly 80/20, but it was very close. The details of these datasets are shown in Table 2.1. In addition, for RQ2, 5-fold cross-validation with stratified group-based splitting at the project level was applied. This approach prevents data leakage [7, 8] by ensuring that all samples from the same project remain within the same fold; thus, a project included in the training set cannot appear in the test set.

Table 2.1: Training and testing datasets. Percentages in parentheses indicate the proportion within the corresponding datasets. Please note that, to enable fair comparisons across all tools and approaches, all accuracy metrics in all RQs were reported using only the test dataset.

Distribution	Dataset	Comments (%)	non-SATD Comments (%)	SATD Comments (%)	Overall SATD %	Projects (%)
Original	Train	38,380 (80.0)	37,902 (80.0)	478 (77.7)	1.2	382 (78.3)
Original	Test	9,614 (20.0)	9,477 (20.0)	137 (22.3)	1.4	108 (22.1)
Deduplicate	Train	26,024 (79.9)	25,621 (80.0)	403 (78.1)	1.5	368 (79.0)
Deduplicate	Test	6,531 (20.1)	6,418 (20.0)	113 (21.9)	1.7	98 (21.0)

2.3.4 Evaluation Metrics

To evaluate SATD detection for RQ2–RQ4, Precision, Recall, and F1-score were used following prior SATD research [25, 76, 77, 101, 102]. Additionally, because SATD comments comprise only 1.28% of the dataset, Matthews Correlation Coefficient (MCC) is also reported, which is more informative for highly imbalanced classification settings. The metrics are calculated as follows.

$$\text{Precision} = \frac{TP}{TP + FP} \quad \text{and} \quad \text{Recall} = \frac{TP}{TP + FN}$$

$$\text{F1-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$\text{MCC} = \frac{(TP \times TN) - (FP \times FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

Here, TP denotes true positives, TN true negatives, FP false positives, and FN false negatives.

2.4 Approach and Results

This section discusses the approach and results of each RQ.

2.4.1 RQ1: What are the different types of SATD in test code?

Motivation. A categorization of SATD will provide a high-level understanding of the types of challenges developers encounter when writing and maintaining tests. This can help reveal which issues are most prevalent or critical in test code, and thus could be prioritized for maintenance and quality improvement efforts. We hypothesize that these categories may differ from those observed in source code, reflecting the unique role and structure of tests within the software development process.

Approach. Manually categorizing SATD comments at a granular level is challenging for several reasons. (i) Prior classifications sometimes distribute the same phenomenon across multiple categories. For instance, Alves et al. [89] classify duplicated code as both code debt and design debt, creating ambiguity during assignment. (ii) Certain comments can encompass more than one category. For example, the comment, `// TODO: tweak the tests when FLINK-13604 is fixed`, simultaneously reflects a dependent defect (awaiting a bug fix) and a requirement (a need to revise the tests). In other cases, interpretation may be subjective, such as with `// An unlikely exception | if this occurs, we can't proceed with the test`, which could reasonably be considered either a defect or a superficial test. (iii) Many SATD comments are terse or lack context (e.g., a standalone TODO), making it difficult to infer the underlying issue without examining surrounding code. (iv). Some comments describe scenarios or concerns not captured by current taxonomies, necessitating extensions to the existing classification scheme. For example, consider the comment `// Probably incorrect - comparing Object[] arrays with Arrays.equals`. This comment cannot be confidently classified as defect debt, nor does it closely align with other existing categories, since the developer expresses uncertainty about whether the issue is truly a defect. Given these challenges—including subjectivity, overlapping categories, and ambiguous content—the categorization of the 615 SATD comments followed established guidelines [18, 89] and joint classification sessions to reach consensus on ambiguous cases. The goal was to preserve diverse categories rather than merge multiple types into broader labels, ensuring

subtle distinctions in SATD were captured.

Initially, the first, second, and last author jointly conducted three online meetings to examine each comment one by one, assessing the nature of the technical debt they represented. This process followed the guidelines proposed by Alves et al. [89], and Bavota and Russo [18], assigning each comment to a specific category based on its characteristics. For each comment, the existing categories from [18, 89] were considered first. If a comment did not fit any existing category, a new category was created to reflect its unique characteristics. The first 200 comments were classified collaboratively by the three authors during those three meetings. However, by the time the first 142 comments had been reviewed, 12 distinct categories had emerged, and no new category was found after reviewing the remaining 58 samples, possibly indicating that a saturation point was reached. The remaining 415 comments were then categorized by the first author independently. In addition to 10+ years of industry experience, this author was also successful in manual labeling in recent times [103, 104].

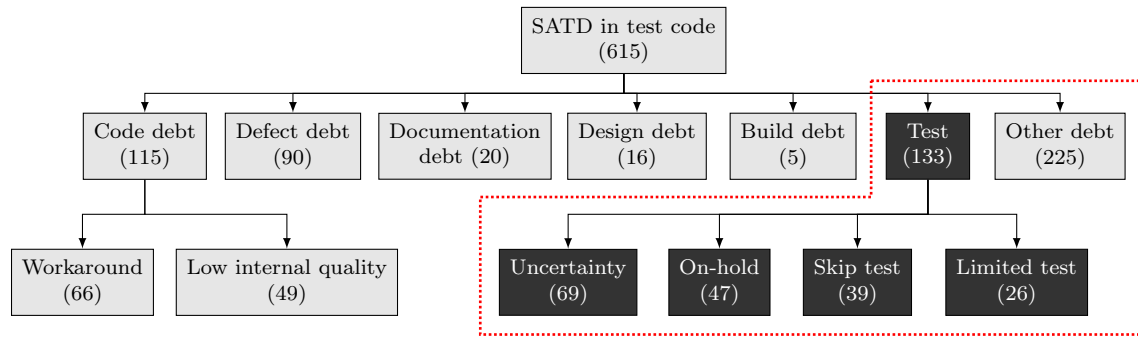


Figure 2.3: Manual categorization of the 615 self-admitted technical debt instances in test code. The categories shown within the dotted red boundary contain SATDs that did not align with any of the existing categories found in the source-code SATD taxonomies compared against. Because some SATD comments were assigned to more than one type, the count shown in a parent node is not necessarily the sum of the counts shown in its descendant nodes.

Results. Figure 2.3 presents the complete hierarchy of these categories and their corresponding sub-categories found in the test code. Four new groups of technical debt were identified, each with unique characteristics, and shown under *Test*. The number of instances

identified in each category is also shown within the respective rectangles. Each example presented in this section is traceable to its original source file, with corresponding links available in the replication package.³

Code debt (115): *“Problems found in the source code which can negatively affect the legibility of the code, making it more difficult to be maintained”* [89]. Two sub-categories of this type of technical debt were identified. The first sub-category, *workaround*, describes intentionally written suboptimal temporary fixes that are justified by the need to meet specific functional or non-functional requirements, often representing a trade-off between code quality and project goals [18]. Representative examples of this sub-category are provided below:

- `//hacky workaround using the toString method to avoid mocking the Ruby runtime`
- `// This is an utter hack but is the only way I could`
- `// this is just a temporary thing but it’s easier to change if it is encapsulated.`
- `// Yes, a static variable is a hack.`
- `// This should use System.getProperty(“os.name”) in a real test.`

The second sub-category, *low internal quality*, refers to deficiencies in the source code, such as poor readability, misuse of programming constructs, or unnecessary complexity [18]. Instance of *low internal quality* are:

- `// TODO(sdh): Print a better name (https://github.com/google/closure-compiler/issues/2982)`
- `// RandomIndexWriter is too slow here:`
- `// TODO: b/359688989 - clean this up`
- `//Ooooooooo this is so ugly`

Defect debt (90): *“Defect debt consists of known defects, usually identified by testing activities or by the user and reported on bug track systems, that the CCB agrees should be*

³https://github.com/SQMLab/technical-debt/blob/main/data/satd_comment_example.csv

fixed, but due to competing priorities, and limited resources have to be deferred to a later time” [89]. Known, unfixed or partially fixed defects were identified, totaling 90 instances. Some examples include:

- `// known bug: time argument is unsigned 32bit in graphics library`
- `// TODO(b/189535612): this is a bug!`
- `//FIXME: fix flaky test`
- `// This test doesn't work on midnight, January 1, 1970 UTC`

Uncertainty (69): Stems from uncertainty or a lack of clarity regarding how to implement, test, or resolve an issue. This type of debt is often marked by open questions, speculative reasoning, or lack of understanding of the expected behavior. Illustrative examples are as follows:

- `// Probably incorrect - comparing Object[] arrays with Arrays.equals`
- `// I don't think doing this will be safe in parallel`
- `// TODO: figure out why I need these`
- `// TODO: Is it correct?`
- `// TODO: This is current behavior, but is it what we want?`

On-hold (47): Refers to cases where the test code relies on external components, such as libraries, APIs, or services, that hinder test execution due to incomplete support or limited functionality. Although in a different context, these types of dependency-induced SATD were also observed by Maipradit et al. [105]. A selection of examples is given below:

- `/** * Setting file last access time does not work on JDK-20 on macOS with the hfs file system, * issue JDK-8298187. */`
- `// NO list support in JDOQL`
`// testData.listTests(store.products, otherStore.products, p);`

- `// C2ID doesn't have a non JS login page :/, so use their API directly // see https://connect2id.com/p/docs/guides/login-page`
- `// mostly waiting for https://github.com/testcontainers/testcontainers-java/issues/3537`

Skip test (39): Arises when certain tests are skipped or explicitly disabled, often due to environmental limitations or unresolved issues. A few examples are:

- `/** * TODO: enable when window is supported. */`
- `// Skip, zstd only supported for Avro 1.9+`
- `// Ignore the following tests because taking a lease requires a real // (not mock) file system store. These tests don't work on the mock.`
- `// Testcase is disabled, as kernel32 ordinal values are not stable. // a library with a stable function j-¿ ordinal value is needed.`
- `// CLDR No Longer uses complex currency symbols. // Skipping this test.`

Limited test (26): Refers to cases where tests provide incomplete or low-value validation, limiting their ability to establish confidence in the behavior under test. Limited tests appeared in two forms. In the first form, only a small, representative portion of the input space is tested, usually for the sake of time efficiency or resource constraints. Examples of this form are:

- `// This only tests for calendar ordering, ignoring the other criteria`
- `// Not testing average length and min/max, as this would make the test less reusable and is not that important to test.`
- `// we do not check if the offset is in the middle of a code point`
- `// FIXME: this test do not cover breaking remark into multiple lines`
- `// Theoretically, the HLL algorithm can count very large cardinality with little relative error rate, // which is less than 2% in Doris. We cost about 5 minutes to test with`

one billion input numbers here // successfully, but in order to shorten the test time, we chose one million numbers.

In the second form, tests exercise code without meaningfully validating its functional correctness. In other words, this is testing for the sake of testing. Examples of this form include:

- // TODO: these tests are too simple since they only replace undefined components
- /** * Sort of a dummy test, as the conditions are perfect. In a more realistic scenario, below, the algorithm needs luck to climb this high. */
- // kinda sorta wrong, but for testing's sake...
- // doesn't make much sense in the context of a list...

Documentation debt (20): *“Refers to the problems found in software project documentation and can be identified by looking for missing, inadequate, or incomplete documentation of any type”* [89]. 20 instances of *documentation* debt were present. Several examples are presented below:

- /** * No Documentation */
- // no JavaDoc
- // TODO: add support for non-existing docs
- /** * todo: describe ChildInfo * * @author Steve Ebersole */
- /** * todo: describe Parent * * @author Steve Ebersole */

Design debt (16): *“Refers to debt that can be discovered by analyzing the source code by identifying the use of practices which violated the principles of good object-oriented design (e.g. very large or tightly coupled classes)”* [89]. 16 instances of design-related technical debt were identified, with representative examples shown below:

- // TODO b/300201845 - encapsulate parents and childs private attributes

- `// TODO(b/123102446): We really like to collapse some of these chained assignments.`
- `// TODO move to core`
- `// Todo: Move this test to SegmentReplicationIndexShardTests so that it runs for both node-node & remote store`
- `/**`
 - `* An implementation of @code OutputStream that should serve as the`
 - `* underlying stream for classes to be tested.`
 - `* In particular this implementation allows to have IOExceptions thrown on demand.`
 - `* For simplicity of use and understanding all fields are public.``*/`

Build debt (5): *“Refers to build related issues that make this task harder, and more time/processing consuming unnecessarily”* [89]. 5 instances of build-related technical debt were identified. Instances of build debt are as follows:

- `//Could fail if CI is to slow and will slow down the CI build; test local`
- `// Continuous build environment doesn’t support non-blocking IO.`
- `// see [GEOS-7835] WPS execution time limits test randomly breaks builds`
- `// Find the async-reset webapp based on common IDE working directories // TODO`
`import webapp as maven artifact`

Other debt (225): This is the most generic form of SATD, and, in general, an SATD was labelled as other only if it did not fit with any other specific type. A total of 225 instances of other debt were identified, with approximately 77% of them beginning with the *TODO* marker—a pattern that is not very common in other categories. Representative examples of other debt are shown below:

- `// TODO this test needs a lot of work`
- `// TODO implement`

- `// TODO implement https://github.com/elastic/elasticsearch/issues/25929`
- `// not implemented yet`

RQ1 summary: The analysis identified 12 SATD types in test code, including 4 new test-specific categories such as skipped and limited tests. Also, for the categories that were found in source code SATD, the distributions are significantly different from the distribution observed here. These findings reveal that test code exhibits unique technical debt patterns not fully captured by existing classifications from application code.

2.4.2 RQ2: Can existing SATD detection tools detect test code SATD?

Motivation. As prior research has shown that self-admitted technical debt (SATD) adversely affects software quality and maintenance efforts [20], the ability to automatically detect SATD is essential for taking timely actions. Yet, it is unknown whether current automated techniques can identify SATD within test code, which is investigated here.

Approach. Seven well-established SATD detection approaches originally developed for source code comments were employed, following the same implementations used in prior comparative work whenever possible.

Pattern. The 62 keyword/phrase patterns summarized by Potdar et al. [10] are used. For each target comment, if any predefined pattern is matched, the comment is labeled as SATD; otherwise, it is labeled as non-SATD. This is a strict rule-based unsupervised method and does not require model training.

MAT. Similar to Pattern, Matches task Annotation Tags (MAT) [76] is unsupervised; however, it differs in two key ways: it is centered on widely used task-annotation tags (e.g., TODO/FIXME-style tags supported by IDEs), and it applies fuzzy matching rather

than strict exact matching, enabling more flexible identification of SATD-like expressions in comments.

SATD Detector. The text-mining based SATD Detector by Liu et al. [21] is included as a supervised classifier that identifies SATD comments from textual features through pre-processing, feature selection, and vector-space representations.

TM. Another text-mining approach by Huang et al. [22] is also included, using the implementation packaged by Guo et al. [76].

NLP. Maldonado et al. [25] formulated SATD detection as an NLP-based text classification task with a Maximum Entropy classifier. As in Guo et al. [76], the available reimplementaion is used (the original code was not released) to ensure reproducibility and a consistent experimental pipeline.

MT-BERT. For the multitask BERT model proposed by Gu et al. [77], the original released implementation is used with default settings reported by the authors. BERT is a bidirectional transformer-based language model that learns contextual representations from text.

DebtHunter. DebtHunter [78], a machine-learning-based SATD detection approach that classifies comments using learned textual features, is also evaluated.

In summary, Pattern, SATD Detector, NLP, TM, and MAT are executed via Guo et al.’s JAR package [76], while MT-BERT and DebtHunter are run using their original implementations [77, 78]. For supervised methods (SATD Detector, NLP, TM, MT-BERT, and DebtHunter), pretrained, retrained, and retrained-with-cross-validation settings are reported where applicable. Precision, Recall, F1-score, and Matthews Correlation Coefficient (MCC) are reported.

Results. The detection results of the seven approaches are summarized in Table 2.2. The pattern-based approach consistently achieved high precision (0.80) but extremely low recall on both the original (0.03) and deduplicated (0.04) test sets, resulting in very low F1-scores (0.06 and 0.07, respectively). Notably, both precision and F1-score are approximately

Table 2.2: SATD detection performance with/without retraining. The unsupervised MAT approach achieved the highest F1-score. Pattern-based methods had high precision but very low recall, and the Text Mining (TM) Method showed low precision but relatively higher recall. We also show the minimum, maximum, and average of all the scores across all the tools and approaches.

Dataset	Method	Retrained?	Cross-Validation?	Precision	Recall	F1-score	MCC
Original	Pattern [10]	–	–	0.80	0.03	0.06	0.15
	MAT [76]	–	–	0.95	0.61	0.75	0.76
	SATD Detector [21]	X	X	0.38	0.58	0.46	0.46
	SATD Detector [21]	✓	X	0.30	0.57	0.40	0.41
	SATD Detector [21]	✓	✓	0.35	0.63	0.44	0.45
	TM [22]	X	X	0.09	0.56	0.15	0.20
	TM [22]	✓	X	0.10	0.86	0.17	0.26
	TM [22]	✓	✓	0.10	0.88	0.18	0.28
	NLP [25]	X	X	0.53	0.54	0.53	0.53
	NLP [25]	✓	X	0.68	0.63	0.65	0.65
	NLP [25]	✓	✓	0.86	0.67	0.75	0.75
	MT-BERT [77]	X	X	0.83	0.63	0.71	0.72
	MT-BERT [77]	✓	X	0.63	0.50	0.56	0.56
	MT-BERT [77]	✓	✓	0.68	0.45	0.54	0.55
	DebtHunter [78]	X	X	0.26	0.58	0.36	0.38
	DebtHunter [78]	✓	X	0.53	0.65	0.59	0.58
	DebtHunter [78]	✓	✓	0.68	0.69	0.68	0.68
	<i>Minimum</i>	–	–	0.09	0.03	0.06	0.15
	<i>Maximum</i>	–	–	0.95	0.88	0.75	0.76
	<i>Average</i>	–	–	0.52	0.59	0.47	0.49
Deduplicate	Pattern [10]	–	–	0.80	0.04	0.07	0.17
	MAT [76]	–	–	0.95	0.56	0.70	0.73
	SATD Detector [21]	X	X	0.37	0.63	0.46	0.47
	SATD Detector [21]	✓	X	0.31	0.61	0.41	0.42
	SATD Detector [21]	✓	✓	0.36	0.65	0.46	0.47
	TM [22]	X	X	0.14	0.49	0.22	0.24
	TM [22]	✓	X	0.28	0.62	0.39	0.40
	TM [22]	✓	✓	0.30	0.63	0.40	0.42
	NLP [25]	X	X	0.49	0.50	0.49	0.48
	NLP [25]	✓	X	0.64	0.58	0.61	0.60
	NLP [25]	✓	✓	0.81	0.62	0.70	0.70
	MT-BERT [77]	X	X	0.83	0.58	0.69	0.69
	MT-BERT [77]	✓	X	0.65	0.43	0.52	0.53
	MT-BERT [77]	✓	✓	0.60	0.41	0.49	0.49
	DebtHunter [78]	X	X	0.34	0.53	0.41	0.41
	DebtHunter [78]	✓	X	0.21	0.65	0.32	0.35
	DebtHunter [78]	✓	✓	0.47	0.69	0.54	0.55
	<i>Minimum</i>	–	–	0.14	0.04	0.07	0.17
	<i>Maximum</i>	–	–	0.95	0.69	0.70	0.73
	<i>Average</i>	–	–	0.50	0.54	0.46	0.48

two to four times lower for the test code dataset than for their original dataset. This outcome highlights the approach’s limited coverage for test code, as it relies on a fixed set

of 62 manually curated patterns obtained through a source code-dominated dataset. The low recall can be attributed, in part, to missing keywords in the manually crafted list, specifically the absence of `TODO` alongside `FIX`, as well as the lack of discernible patterns in many comments.

The MAT approach achieved the best overall performance across both datasets without any training. It consistently yielded high precision (0.95) and achieved the highest F1-scores on both the original (0.75) and deduplicated (0.70) datasets, with corresponding recall values of 0.61 and 0.56. Notably, both MAT and the NLP approach achieved the highest F1-scores, with NLP attaining this performance under the cross-validation setting. Nevertheless, compared to prior work, the F1-score observed in this evaluation is 2% lower.

The two text-mining approaches exhibit notably different performance profiles on test code comments. Compared with Huang et al.’s TM approach, SATD Detector consistently achieves much higher precision, F1-score, and MCC on both datasets, while maintaining competitive recall. However, SATD Detector remains clearly below the retrained NLP approach, which attains substantially stronger balanced performance, indicating that SATD Detector provides better coverage-oriented performance than TM but not the same overall effectiveness as NLP. Meanwhile, the TM approach produced consistently low precision and F1-scores across all settings. Similar to the NLP pretrained model, TM failed to recognize `TODO`, although it correctly identified `FIXME`, and produced numerous false positives from patterns such as issue IDs (e.g., `// DR00LS-155`), copyright notices, and Javadocs with author annotations. After retraining, TM corrected many of these patterns but still misclassified a substantial number of irrelevant comments as SATD. Nevertheless, its recall was comparable to that of the NLP and MAT approaches, and was even the highest for the retrained cross-validation setting on the original dataset. Compared to Huang et al.’s [22] reported F1-score of 0.737 for source code, its performance on test code is notably poor.

For both datasets, the retrained NLP model achieved roughly 12% higher F1-scores than the pretrained model, producing balanced precision and recall. On the pretrained model,

many comments containing the two major keywords, `TODO` and `FIXME`, were missed, and the misidentified comments generally lacked clear patterns. After retraining, the model successfully learned `TODO` but still failed to capture `FIXME`. Notably, most of the remaining misclassified comments were Javadocs describing bug-related tests that include bug ID (e.g., `@bug 8077931`). Compared to Guo et al.’s [76] result on source code, the best F1-score in this evaluation after retraining is about 5% lower.

The MT-BERT approach exhibited strong performance on pretrained, achieving F1-scores of 0.71 and 0.69 on the original and deduplicated test datasets, respectively. This indicates that the pretrained multitask model generalizes reasonably well to test code comments. However, retraining MT-BERT on the test code comment datasets resulted in a noticeable performance degradation, primarily due to reduced recall, which led to lower F1-scores. Manual inspection revealed that MT-BERT attained relatively high precision by correctly identifying SATD comments containing explicit debt-related keywords; nevertheless, it occasionally failed to detect when common keywords such as `TODO` were present, and similar to MAT, it struggled with comments lacking explicit keyword signals, leading to comparable recall between the two approaches. Overall, the pretrained version performed better in these cases, likely because it was trained on a larger and more diverse set of SATD patterns, enabling it to better capture keyword-based and recurring SATD expressions. DebtHunter improved after retraining, achieving stronger recall and F1-score under cross-validation. However, its lower precision kept its overall performance below the best MAT and NLP results.

The overall poor to moderate performance indicates that methods originally developed based on the characteristics of source code fail to fully capture the distinctive properties of test code. This finding reinforces the notion that SATD instances in test code differ substantially from those in source code. Although retraining sometimes improved the performance, even in the best scenarios, results remained 2–5% lower than those on the source code. These findings highlight the somewhat poor transferability of source-trained models and show that

retraining alone is not sufficient.

RQ2 summary: Although the NLP and MAT approaches achieve the highest accuracy in detecting SATD in test code, their F1-scores remain relatively modest at around 0.75. This limitation motivated the investigation of different LLMs for detecting SATD in test code, particularly because prior research has demonstrated the strong performance of LLMs across various software engineering tasks, including the detection of SATD in source code.

2.4.3 RQ3: Can open-source LLM detect SATD in test code?

Motivation. Although SATD often contains keywords such as *TODO*, many comments do not follow such patterns. In these cases, even sophisticated pattern-based methods (e.g., TM, NLP) fail to detect SATD, highlighting the need for approaches that capture semantic meaning. Consequently, Large Language Models (LLMs) are investigated, given their strong capability in understanding both natural language and source code [106]. Recent studies have also found promising results with LLMs for detecting source code SATD [24, 88], outperforming conventional approaches.

Approach. Encouraged by the success of SATD detection [24] of the Flan-T5 model series, these models were investigated on test code SATD with few-shot configurations of 0, 2, and 4 shots. These models are T5-SMALL, T5-BASE, T5-LARGE, T5-XL, and T5-XXL, with numbers of parameters 60M to 11B. The experiments were run on an HPC cluster with Intel 8570 CPUs (2.1 GHz) and multiple NVIDIA H100 GPUs. Four different prompt templates were tested: No Keyword, MAT, Jitterbug, and GPT-4. These were adopted from Sheikhaei et al. [24] to enable direct comparison with their results.

Additionally, a new prompt template developed mainly for RQ4 was added, as illustrated in Figure 2.4. Existing research highlights that LLMs are highly sensitive to the structure and phrasing of prompts, with their performance varying considerably depending on how

Instruction

You are a Code Analysis Expert specialized in detecting Self-Admitted Technical Debt (SATD) in Java test code comments.

SATD refers to comments where developers acknowledge that the current test implementation is incomplete, suboptimal, or relies on a compromise that should be addressed in the future. These admissions often appear as markers such as TODO or FIXME, or as notes about unresolved issues, temporary fixes, workarounds, hacks, performance limitations, use of deprecated APIs, unsupported features, poor design choices, skipped tests, or uncertain functionality. However, comments that only describe expected behavior of test code are not SATD unless there is additional information indicating the need for future improvement.

Think step by step and assign the label of yes or no for each given test code comment.

Few-Shot Examples

Comment: <<example comment>>

Answer: <<explanation>> The answer is <<label>>.

.....

Query

Comment: <<comment for the LLM to detect>>

Answer:

Figure 2.4: LLM prompt template for detecting SATD comment.

tasks, instructions, and examples are articulated [107]. To ensure greater reliability and adaptability, a modular prompt template was developed, comprising three components, in accordance with established prompt design frameworks [108, 109]. The first, the Instruction, is organized into three parts: (i) a role-based persona that anchors the model’s behavior and enhances response quality [110]; (ii) a contextual and task-specific description that clarifies the model’s objective [108]; and (iii) explicit output formatting guidelines that reduce ambiguity and improve consistency in responses. The second, Few-Shot Examples, varies in number across experiments to assess performance sensitivity. Finally, the third, Query, contains the target comment that the LLM is required to analyze and classify.

Results. Table 2.3 reports the results for the T5 model family, ranging from T5-SMALL to T5-XXL. The findings indicate that the Flan-T5 series LLMs are unable to consistently detect SATD in test code, with their effectiveness varying depending on the prompting strategy, shots, and model size. In most cases, the smaller models—T5-Small, T5-Base, and T5-Large—failed to detect SATD, resulting in zero recall and, consequently, zero precision.

Between the two largest models, T5-XL consistently achieved higher precision than T5-XXL, often reaching near-perfect precision (e.g., 1.0) with keyword-based prompts (MAT and Jitterbug). However, this high precision generally came at the expense of very low recall. The New prompt performed competitively, achieving a maximum F1-score of 0.54. In contrast, the No Keyword and GPT-4 prompts yielded poor results, with many zero-shot and few-shot runs failing to detect any SATD ($F1 = 0.0$). Meanwhile, the T5-XXL models demonstrated higher recall, frequently exceeding 0.8, but with substantially lower precision (ranging from 0.10 to 0.42).

When comparing the F1-scores of T5-XXL across different prompt variations with those reported by Sheikhaei et al. [24], similar overall trends are observed; however, the scores in this evaluation are approximately 30% lower. Overall, the MAT-based keyword prompts achieved the highest F1-score (0.68). This strong performance is largely due to the high prevalence of MAT-suggested keywords—particularly *TODO*—in SATD comments and their

Table 2.3: Precision (P), Recall (R), F1-score (F1), and Matthews Correlation Coefficient (MCC) for detecting SATD comments using Flan-T5 series open-source LLMs across 0-, 2-, and 4-shot settings.

Dataset	Prompt	Model	0-shot				2-shots				4-shots			
			P	R	F1	MCC	P	R	F1	MCC	P	R	F1	MCC
Original	New	T5-SMALL	0.00	0.02	0.01	-0.02	0.00	0.00	0.00	-0.01	0.00	0.00	0.00	-0.01
		T5-BASE	0.01	0.31	0.02	-0.03	0.02	0.51	0.03	0.01	0.01	0.45	0.03	0.01
		T5-LARGE	0.03	0.96	0.05	0.10	0.04	0.91	0.07	0.14	0.04	0.74	0.07	0.12
		T5-XL	0.29	0.86	0.43	0.49	0.38	0.79	0.51	0.54	0.47	0.64	0.54	0.54
		T5-XXL	0.10	0.98	0.18	0.29	0.12	0.96	0.21	0.32	0.11	0.95	0.20	0.31
	No Keyword	T5-SMALL	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		T5-BASE	0.00	0.03	0.01	-0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-0.00
		T5-LARGE	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		T5-XL	0.33	0.01	0.01	0.05	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		T5-XXL	0.10	0.66	0.18	0.24	0.19	0.50	0.28	0.29	0.17	0.50	0.25	0.28
	GPT 4	T5-SMALL	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		T5-BASE	0.01	0.04	0.01	-0.03	0.00	0.00	0.00	-0.00	0.00	0.00	0.00	-0.02
		T5-LARGE	0.24	0.04	0.06	0.09	0.29	0.01	0.03	0.06	0.67	0.01	0.03	0.10
		T5-XL	1.00	0.04	0.07	0.19	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		T5-XXL	0.20	0.87	0.32	0.40	0.18	0.82	0.30	0.37	0.18	0.80	0.30	0.36
	Jitterbug	T5-SMALL	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		T5-BASE	0.00	0.03	0.01	-0.03	0.00	0.00	0.00	-0.00	0.00	0.00	0.00	-0.01
		T5-LARGE	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		T5-XL	1.00	0.47	0.64	0.68	1.00	0.11	0.20	0.33	1.00	0.08	0.15	0.28
		T5-XXL	0.28	0.83	0.42	0.47	0.27	0.82	0.41	0.46	0.30	0.80	0.43	0.48
	MAT	T5-SMALL	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		T5-BASE	0.00	0.04	0.01	-0.03	0.00	0.00	0.00	-0.00	0.00	0.00	0.00	-0.00
		T5-LARGE	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		T5-XL	0.99	0.52	0.68	0.71	1.00	0.13	0.23	0.36	1.00	0.12	0.21	0.34
		T5-XXL	0.33	0.80	0.47	0.50	0.40	0.78	0.53	0.55	0.42	0.74	0.54	0.55
Deduplicate	New	T5-SMALL	0.01	0.03	0.01	-0.02	0.00	0.00	0.00	-0.01	0.00	0.00	0.00	-0.01
		T5-BASE	0.02	0.36	0.04	0.01	0.02	0.55	0.04	0.03	0.02	0.52	0.04	0.03
		T5-LARGE	0.03	0.95	0.05	0.09	0.04	0.89	0.08	0.14	0.05	0.76	0.09	0.14
		T5-XL	0.26	0.85	0.40	0.46	0.36	0.76	0.49	0.51	0.46	0.65	0.54	0.53
		T5-XXL	0.10	0.97	0.18	0.28	0.11	0.97	0.20	0.31	0.11	0.96	0.20	0.30
	No Keyword	T5-SMALL	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		T5-BASE	0.01	0.04	0.02	-0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	-0.00
		T5-LARGE	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		T5-XL	0.00	0.00	0.00	-0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		T5-XXL	0.11	0.74	0.20	0.26	0.22	0.58	0.32	0.34	0.21	0.57	0.31	0.33
	GPT 4	T5-SMALL	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		T5-BASE	0.01	0.04	0.02	-0.01	0.00	0.00	0.00	-0.00	0.00	0.00	0.00	-0.01
		T5-LARGE	0.25	0.04	0.07	0.10	0.29	0.02	0.03	0.07	0.67	0.02	0.03	0.11
		T5-XL	1.00	0.04	0.07	0.19	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		T5-XXL	0.19	0.87	0.31	0.39	0.18	0.87	0.29	0.37	0.18	0.83	0.30	0.37
	Jitterbug	T5-SMALL	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		T5-BASE	0.01	0.02	0.01	-0.01	0.00	0.00	0.00	-0.00	0.00	0.00	0.00	-0.01
		T5-LARGE	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		T5-XL	1.00	0.40	0.57	0.63	1.00	0.12	0.22	0.35	1.00	0.09	0.16	0.29
		T5-XXL	0.28	0.82	0.42	0.47	0.26	0.81	0.39	0.44	0.28	0.79	0.41	0.45
	MAT	T5-SMALL	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		T5-BASE	0.01	0.03	0.01	-0.01	0.00	0.00	0.00	-0.00	0.00	0.00	0.00	-0.00
		T5-LARGE	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
		T5-XL	1.00	0.46	0.63	0.68	1.00	0.15	0.26	0.39	1.00	0.13	0.23	0.36
		T5-XXL	0.34	0.79	0.47	0.50	0.40	0.76	0.52	0.54	0.40	0.72	0.52	0.53

absence in non-SATD comments within the test dataset. For instance, using the MAT approach, the model correctly identified 71 SATD comments out of 137 with the Flan-T5-XL (zero-shot) setup but failed to detect the remaining 66, even when other obvious indicators (e.g., yuck, workaround) were present. A clear example of a missed case is the comment “*// kinda sorta wrong, but for testing’s sake...*”, which expresses technical debt but lacks any MAT-suggested keywords.

These findings indicate that the choice of prompt has a substantial impact on model performance. Increasing the number of shots (0→2→4), however, did not consistently improve results; this trend was also observed by Sheikhaei et al. [24]. In both studies, the best performance was achieved under zero-shot settings, as demonstrated by T5-XL with MAT-based prompts in these experiments.

RQ3 summary: The results indicate that open-source LLMs (Flan-T5 series) struggle to reliably detect SATD in test code across different prompt types and shot settings. Although the best F1-score reached 0.68 using the MAT prompt, the overall performance remains inconsistent and relatively poor compared to the existing tools (RQ2), suggesting that these models have a limited ability to understand and generalize SATD patterns in test code.

2.4.4 RQ4: Can proprietary LLM detect SATD in test code?

Motivation. Motivated by the unexpectedly poor performance of open-source LLMs observed in RQ3, this section extends the investigation to proprietary LLMs. Previous research consistently reports that proprietary models tend to outperform open-source counterparts across a range of standard benchmarks, including those that involve code understanding and generation tasks [79, 111, 112].

Approach. Two major proprietary LLM families—Gemini and GPT—were used to assess their effectiveness in detecting SATD in test code. For Gemini, the latest Flash

variants of Gemini 2.0 and 2.5 were used, and for GPT, three GPT-5 models (nano, mini, and standard) were tested to capture diverse reasoning capabilities and model sizes. Each model was tested under the few-shot configurations of 0, 2, and 4 shots. All the experiments were conducted using the prompt shown in Figure 2.4.

Table 2.4: Precision (P), Recall (R), F1-score (F1) and Matthews Correlation Coefficient (MCC) for detecting SATD comments using GPT and Gemini across 0-, 2-, and 4-shot settings.

Dataset	Model	0-shot				2-shots				4-shots			
		P	R	F1	MCC	P	R	F1	MCC	P	R	F1	MCC
Original	gpt-5-nano	0.12	0.93	0.20	0.31	0.18	0.91	0.30	0.39	0.19	0.91	0.31	0.40
	gpt-5-mini	0.23	0.95	0.36	0.45	0.25	0.93	0.39	0.47	0.26	0.96	0.40	0.48
	gpt-5	0.27	0.95	0.42	0.50	0.34	0.93	0.49	0.55	0.32	0.94	0.47	0.54
	gemini-2.5-flash	0.15	0.99	0.26	0.36	0.17	1.00	0.28	0.39	0.12	0.99	0.21	0.33
	gemini-2.0-flash	0.24	0.88	0.37	0.44	0.15	0.98	0.26	0.37	0.12	0.99	0.21	0.32
Deduplicate	gpt-5-nano	0.11	0.92	0.20	0.30	0.19	0.91	0.31	0.39	0.17	0.89	0.29	0.38
	gpt-5-mini	0.22	0.94	0.35	0.44	0.24	0.94	0.39	0.47	0.24	0.96	0.38	0.46
	gpt-5	0.27	0.96	0.42	0.49	0.31	0.92	0.46	0.52	0.30	0.93	0.45	0.51
	gemini-2.5-flash	0.15	0.99	0.26	0.36	0.17	1.00	0.29	0.39	0.12	0.99	0.21	0.31
	gemini-2.0-flash	0.24	0.88	0.38	0.45	0.15	0.98	0.27	0.37	0.11	0.99	0.20	0.31

Results. The experimental results are summarized in Table 2.4. Proprietary models achieved consistently high recall, frequently exceeding 0.90 and occasionally reaching 1.0 (e.g., Gemini-2.5-Flash under 2-shot). However, this high recall was accompanied by very low precision, particularly for Gemini models, where precision often ranged between 0.11 and 0.24. Among GPT models, the standard GPT-5 demonstrated the most balanced performance, attaining the highest F1-scores under 2-shot settings on both the original (0.49) and deduplicated (0.46) datasets. The GPT-5 mini and nano variants performed moderately well, outperforming Gemini in terms of precision and F1, but lagging behind GPT-5 standard. Increasing the number of shots did not consistently improve performance, as was seen with the Flan-T5 models as well.

Compared to the Flan-T5 series (RQ3), proprietary models exhibit distinct performance patterns. GPT-5 models maintain high recall, unlike T5-XL, which achieves strong precision primarily with MAT or Jitterbug prompts but suffers from limited recall. GPT-5

provides more stable SATD detection across shot settings compared to the Flan-T5 series. Nonetheless, Flan-T5-XL with MAT prompts achieved the highest F1-score (0.68).

RQ4 summary: In general, proprietary LLMs, particularly GPT-5, outperform open-source models in recall. However, their much lower precision indicates a tendency to over-predict SATD instances. These results suggest that while proprietary models may offer stronger contextual reasoning to exhibit high recall, significant improvements are needed to address their limitations with precision.

2.5 Discussion

SATD can significantly hinder software maintenance activities. Understanding the nature of SATD and its detection is, therefore, critical for effective mitigation. The landscape of SATD in test code is investigated here—an area largely overlooked by previous research, despite the critical role of testing in software development and the well-documented impact of SATD on software maintainability. The types of SATD that appear in test code (RQ1) were examined, and 12 types were found, where 4 of them were new—not found in other source code-based studies. The effectiveness of the existing tools (RQ2), open-source LLMs (RQ3), and proprietary LLMs (RQ4) in identifying these SATD comments was then assessed. Existing tools, particularly MAT, were able to detect SATD with high precision (0.95), but their recall (≤ 0.61) remains limited. Meanwhile, open-source LLMs exhibited inconsistent and, in some cases, unexpectedly poor performance across model sizes and prompt variations. In the best cases, such as Flan-T5-XL with the MAT prompt, high precision but low recall was observed. In contrast, proprietary LLMs exhibited very high recall but very low precision.

While a complete systematic qualitative analysis of the inaccurate detection of test code SATD by all studied tools and approaches is left as future work, an initial effort was still made to understand the root causes of these failures. To this end, the first author, who has over 10 years of industry experience, manually analyzed the best-performing configuration in

each of the three approach families: MAT among existing tools, Flan-T5-XL with the MAT prompt in the zero-shot setting among open-source LLMs, and GPT-5 in the 2-shot setting among proprietary LLMs. For the best-performing existing approach, MAT, the first author manually analyzed all 50 missed SATD comments in the deduplicated test set. They found that, although many of these false negatives do express technical debt, they do so through test-specific wording rather than the source-code SATD cues that existing approaches were largely designed around. Representative examples include the following:

- `// inline function cannot be recursive // note: mutual recursion is not supported either, but it is not tested due to the forward declaration limitation above`
- `// Disabled until replace supports external destinations. To be enabled after that point.`
- `// Not testing average length and min/max, as this would make the test less reusable and is not that important to test.`

These examples indicate that lower recall is concentrated in test-specific wordings or expressions that are common in test SATD comments. Even when comments contain words such as `disabled`, `deactivated`, or `skip`, these expressions function as test-specific cues rather than canonical source-code SATD markers such as `TODO` or `FIXME`. As a result, keyword- and pattern-dependent approaches can preserve high precision while still missing a large portion of genuine SATD comments.

For open-source LLMs, the same author manually analyzed all 61 SATD comments missed by Flan-T5-XL in the MAT-based zero-shot setting. Again, many missed comments relied on test-specific phrasing rather than conventional SATD keywords. Although these models are more flexible than strict rule-based techniques, their strongest results still come from keyword-oriented prompts such as MAT and Jitterbug, suggesting that they remain strongly anchored to lexical cues inherited from source-code SATD. Examples include:

- `// this test has been disabled because of timeout issues. // see JDK-8006746`

- `// Deactivated due to https://github.com/bazelbuild/bazel/issues/9380.`
- `// skip test, as it needs historical time zone info`

Interestingly, the proprietary LLMs exhibited the opposite behavior: very high recall but very low precision. To understand their lower precision, 146 of the 233 non-SATD comments misclassified as SATD (95% confidence level) by GPT-5 in the 2-shot setting were randomly sampled and manually analyzed. In these cases, the model often overinterpreted generic task-oriented, cautionary, or context-setting comments as technical debt. Representative false positives include:

- `//Finish pending tasks`
- `// Load it dynamically because Hibernate won't be on the classpath when we're testing with Eclipselink`
- `// Wait for a second so that the auth is synced, to avoid flakiness`

These examples help explain why proprietary LLMs achieve high recall but low precision. Such models appear to use broader semantic reasoning, which allows them to recover SATD comments that keyword-dependent approaches miss. However, the same behavior can cause them to overinterpret ordinary explanatory comments as debt. In the first example, the phrase **pending tasks** can be read as unresolved work, even though the comment merely describes the purpose of the following statement. In the second example, the model may interpret the dependency-related wording as evidence of an architectural limitation or workaround, although the comment only explains a test setup decision under a specific classpath condition. In the third example, the phrasing resembles a temporary workaround for unstable behavior, which can sound like admitted debt, but the comment is still describing test synchronization logic rather than explicitly admitting postponed maintenance. Taken together, these cases show that broader semantic reasoning improves recall, but it can also weaken discrimination between SATD and non-SATD comments when operational,

dependency-related, or test-stabilization comments resemble debt-related language. More generally, the quantitative and qualitative results indicate that SATD in test code differs from source-code SATD not only in category but also in how they are generally expressed. Reliable detection of SATD in test comments will therefore likely require models that learn test-specific cues in context, rather than relying only on source-code SATD keywords or on overly broad semantic interpretations.

Future work can explore additional open-source models, incorporate richer contextual information (e.g., surrounding code), and train or fine-tune models to improve detection accuracy. In particular, future training data and prompts should explicitly capture test-specific SATD expressions and keywords such as `disabled`, `deactivated`, and `skip`, rather than relying mainly on source-code-oriented cues such as `TODO` and `FIXME`. Including such test-specific cues, together with hard negative examples that use similar operational language without actually admitting debt, may improve both recall and precision. When such accurate detection is possible, future work can investigate the evolution of test methods with SATD [20, 45] to understand their long-term maintenance impact (e.g., change- and bug-proneness of SATD test methods compared to non-SATD test methods). The manually curated dataset of test code SATD can support future research on automated test code SATD detection, classification, and the impact of test code SATD in long-term software maintenance.

2.6 Threats to Validity

Construct validity. The SATD detection and classification were completely based on manual analysis. Since manual interpretation is inherently susceptible to personal bias and subjectivity, multiple reviewers were involved to mitigate this risk.

Internal validity. Internal validity is hampered by the selection of LLMs. However, the open-source LLMs were selected based on their excellent performance in a similar task.

While selecting the proprietary LLMs, the focus was on the prominent Gemini and GPT models that exhibited good performance in many software engineering tasks [113–115].

External validity. To enhance the generalizability of the findings, 1,000 open-source projects spanning a variety of domains were analyzed. However, all selected projects were written in Java. As a result, the findings may not generalize to closed-source projects or software developed in other programming languages. Also, despite analyzing a large number of comments (50K), the relatively low number of SATD comments in the manually reviewed dataset remains a limitation; expanding this dataset could enhance the robustness of the findings.

Conclusion validity is hampered by all the aforementioned threats.

2.7 Conclusion

A large-scale empirical investigation of Self-Admitted Technical Debt (SATD) in test code is presented. Through manual analysis of 50,000 comments from 1,000 open-source Java projects, 615 SATD instances were identified and further categorized into 12 types, including four newly proposed ones. The findings reveal that SATD in test code exhibits distinct characteristics compared to SATD in source code. The evaluation of SATD detection approaches further demonstrates that existing tools achieve high precision but suffer from low recall, open-source LLMs display inconsistent performance across models and prompt configurations, and proprietary LLMs attain higher recall at the expense of precision. These insights can inspire future research and facilitate the development of more robust, and accurate techniques for the automated detection and classification of SATD. Ultimately, such advancements have the potential to support practical adoption in industry, enabling developers to better manage technical debt and reduce long-term maintenance overhead.

Part II

Method-History Generation and Test Method Evolution

Chapter 3

HistoryFinder: Accurate Oracle Construction and Method-History Generation

This chapter begins Part II and presents the second contribution of this thesis: improving the reliability of method-level history generation. Many software-evolution studies rely on accurate histories of individual methods, including studies of change-proneness, bug-proneness, technical debt, and test-code evolution. In this thesis, such histories are also required for Chapter 4, where test-method history construction is evaluated before test-method evolution is studied. Therefore, this chapter addresses two related methodological problems: constructing reliable method-history oracles and improving tool support for method-history generation.

This chapter is based on the following manuscript:

- S. Islam, A. Aowal, M. S. Uddin, and S. Chowdhury, “HistoryFinder: Advancing Method-Level Source Code History Generation with Accurate Oracles and Enhanced Algorithm,” manuscript accepted for publication in the proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE), Rio de Janeiro, Brazil, 2026.

Abstract

Reconstructing a method’s change history efficiently and accurately is critical for many software engineering tasks, including maintenance, refactoring, and comprehension. Despite the availability of method history generation tools such as *CodeShovel* and *CodeTracker*, existing evaluations of their effectiveness are limited by inaccuracies in the ground truth oracles used. Two new oracles—the corrected *CodeShovel* oracle and a newly developed *HistoryFinder* oracle—are systematically constructed by combining automated analysis with expert-guided manual validation. *HistoryFinder* is also introduced as a new method history generation tool designed to improve not only the accuracy and completeness of method change histories but also to offer competitive runtime performance. The evaluation across 400 methods from 40 open-source repositories shows that *HistoryFinder* consistently outperforms *CodeShovel*, *CodeTracker*, *IntelliJ*, and **Git-based** baselines in terms of precision, recall, and F1 score. Moreover, *HistoryFinder* achieves competitive runtime performance, offering the lowest mean and median execution times among all the research-based tools. While **Git-based** tools exhibit the fastest runtimes, this efficiency comes at the cost of significantly lower precision and recall—leaving *HistoryFinder* as the best overall choice when both accuracy and efficiency are important. To facilitate adoption, a web interface, CLI, and Java library are provided for flexible usage.

3.1 Introduction

Source code change history is valuable to both software practitioners and researchers. Practitioners leverage source code history for tasks such as code reviews, expert identification, and developer onboarding [45]. Researchers, on the other hand, use code history for a variety of tasks, including bug prediction [8, 116], code clone management [46], entity name recommendation [47], security vulnerability detection [117], code review [45], authorship attribution [48, 49], and software maintenance effort estimation [9, 20, 118]. To extract this historical information, researchers primarily relied on version control systems (VCS). However, traditional VCSs like `Git` mainly focus on file-level change tracking, which lacks the granularity needed for studies at finer levels. In practice, understanding maintenance activities at lower granularities—particularly at the method level—is important to both researchers and practitioners [8, 20, 45]. However, tracking method-level change history is challenging, as methods may be renamed or moved across files during refactoring and other development activities. These challenges and practical demands have driven the development of specialized tools for generating method-level change histories [45, 59, 60].

Several approaches have been proposed to recover method-level code evolution, particularly from `Git` repositories. Hata *et al.* [58] introduced *Historage*, a tool that enables method-level change tracking by restructuring repositories to store individual methods in separate files. This approach was later refined in *FinerGit* [59], which enhanced the accuracy of detecting small method changes. However, both tools require extensive preprocessing [45] of the codebase before any method-level history can be generated. This introduces additional storage overhead and preprocessing time, making them impractical for scenarios where fast and on-demand change history extraction is needed.

To mitigate the need for upfront preprocessing, Grund *et al.* [45] introduced *CodeShovel* (ICSE 2021), a tool that tracks method-level change history on demand using a lightweight string similarity algorithm. *CodeShovel* was reported to be both accurate and efficient, based on evaluation using a human-generated change history oracle of 200 Java methods.

Unfortunately, Jodavi *et al.* [60] later identified inaccuracies in this oracle. They corrected the oracle and introduced a more robust and refactoring-aware tool, *CodeTracker* (ESEC/FSE 2022), which significantly outperformed *CodeShovel* in terms of precision and recall.

A concern remains, however, that the internal threats present in the human-generated *CodeShovel* oracle may have carried over into the *CodeTracker* oracle, which was largely derived from *CodeShovel* and later corrected by a separate group of researchers, thereby introducing its own set of internal threats. Such inaccuracies in recorded change histories are particularly problematic, as tools validated by these oracles form the foundation for research on method-level maintenance analysis and modeling [8, 9, 20, 65, 119]. Moreover, the accuracy of tools evaluated against these less reliable oracles remains uncertain, raising concerns about their practical usefulness to software practitioners. Low precision in identifying method change commits can result in excessive false positives, while low recall may lead to missing critical changes—both of which can harm the usefulness of change history tracking in real-world scenarios.

To address the limitations of human-generated change histories, a more systematic, semi-automated approach was adopted to construct two new oracles for method-level change history. Rather than relying solely on manual annotation, a tool was developed to collect method-change commits for a given Java method using multiple state-of-the-art tools—namely, *CodeShovel*, *CodeTracker*, *GitLog* (*GitFuncName* and *GitLineRange*), and *IntelliJ*. The tool aggregates method change commits collected by all sources, based on the observation that this combined approach achieves significantly higher recall than any individual tool. This union set was then manually verified by two expert software engineers, who removed false positives to ensure high precision and added any relevant method change commits missed by all tools. The result is two high-quality oracles that combine the strengths of automated detection and expert validation. As shown later in the analysis (**RQ1**), the previously used oracles from *CodeShovel* and *CodeTracker* contain notable inaccuracies—confirming concerns about the validity of prior evaluations that relied on those oracles.

In addition to providing the new oracles, a novel, heuristic-based, method-level history generation algorithm and tool called *HistoryFinder* is introduced. It also addresses several of *CodeShovel*'s technical limitations. For instance, unlike *CodeShovel*, *HistoryFinder* captures changes introduced in merge commits, as well as modifications to JavaDocs, annotations, and code formatting.

HistoryFinder is then compared with other method-level change-history tracking tools in terms of precision, recall, and runtime using the two newly developed oracles (**RQ2** and **RQ3**). Overall, the research-oriented tools—*HistoryFinder*, *CodeShovel*, and *CodeTracker*—consistently demonstrate substantially higher accuracy than other alternatives such as *GitLog* and *IntelliJ*. *HistoryFinder* generally achieves the highest precision and recall across all test oracles, while also offering competitive runtime performance. Although *GitLog* is the fastest at reconstructing method histories, it suffers from significantly lower precision and recall, making *HistoryFinder* the most balanced option in terms of accuracy and efficiency. To support broader adoption, a **GUI-based** tool is also provided that allows researchers and practitioners to generate method-level change histories using any of the state-of-the-art tools.

The contribution of this chapter can be summarized as follows.

- It offers two systematically developed method-level change history oracles—consisting of 400 Java methods. Building these oracles with manual verification required approximately 300 hours from each of the two participating authors, excluding the time for building the necessary tools. These more accurate new oracles can be instrumental for future research on developing more method-level change history generation tools.
- It introduces a new algorithm and tool, *HistoryFinder*, that addresses the limitations of the *CodeShovel* tool.
- It conducts a thorough evaluation of the state-of-the-art method tracking tools against two accurate oracles to assess their relative performance in terms of precision, recall,

F1 score, and runtime efficiency.

- It provides a user-friendly GUI-based tool that integrates all the state-of-the-art method-tracking tools (excluding *IntelliJ*), allowing users to analyze and generate method histories for any arbitrary Java method from a **Git-based** project.

The two oracles and the GUI-based history generation tool are publicly shared.¹ While this **GUI-based** tool can be used by both practitioners and researchers, the shared repository also shows how to use the new *HistoryFinder* tool from the command line for enabling research in *mining software repositories*.

3.2 Background and Related Work

This section discusses earlier research that leveraged code change history, especially at the method level, to analyze and build maintenance models. It then discusses research that focused on building history construction tools.

3.2.1 Research on Code Change History and Maintenance

Software maintenance models commonly target the prediction of changes and bugs, as components that undergo frequent modifications or are tied to bug fixes are often harmful [9, 120–122]. Early identification of such components can substantially lower future maintenance effort and cost [9, 123–125]. Leveraging various code metrics, Koru *et al.* [123] built tree-based models to forecast change-prone classes. In a similar vein, Romano and Pinzger [5] applied code metrics to predict change-prone **Java** interfaces. Khomh *et al.* [6] demonstrated that certain types of code smells can also signal classes likely to change. Bug prediction models pursue a similar objective to *change-proneness* prediction models: identifying code elements—such as classes or files—that are more likely to contain bugs in future versions [126–131]. These studies, however, primarily focused on the class or file level, which

¹*HistoryFinder*: <https://github.com/SQMLab/HistoryFinder>

has been reported as less useful by both practitioners and researchers [7, 8, 45, 84]. For example, finding bugs in a class or file can be difficult due to its large size. Consequently, method-level change and bug prediction models have gained traction in recent years.

Method-level bug prediction has been studied by Giger *et al.* [84], Mo *et al.* [132], Ferenc *et al.* [133], Shippey *et al.* [134], Hata *et al.* [135], Menzies *et al.* [136], Pascarella *et al.* [7], and Chowdhury *et al.* [8]. These studies heavily relied on constructing method-level change histories—either to build labeled datasets or to derive change metrics used as input features for machine learning models. For instance, Mo *et al.* [132] employed metrics such as the number of added and deleted lines within a method to predict its bug-proneness, a strategy also adopted by Pascarella *et al.* [7]. Chowdhury *et al.* [8] used method-level change history to identify bug-fixing commits and label methods as buggy or non-buggy. The accuracy of the tools used to construct method-level change histories is therefore critical, as inaccuracies can lead to erroneous change metrics and incorrect bug labels.

3.2.2 Research on History Construction

Earlier studies on method-level tracking in `Git` used separate files to record method changes alongside `Git`'s file-level history. After extensive preprocessing of the entire codebase, this approach enables the use of `git log` to retrieve method histories. Hata *et al.* [58] introduced this technique with *Historage*, which tracks changes to constructors, methods, and fields. However, Higo *et al.* [59] found that *Historage* struggles to track small methods when renamed or moved across files. To overcome these issues, they proposed *FinerGit*, which represents methods with one token per line and applies heuristics for change comparison. Still, the preprocessing is time-intensive and incurs significant storage overhead—an issue common across similar tools [137–140]. Other tools [141, 142] analyze changes between two versions but fall short of supporting origin analysis required by developers [45] and researchers [9].

Recent studies have proposed solutions that eliminate the need for upfront repository preprocessing and avoid tracking methods at each `Git` commit [45, 60]. Grund *et al.* [45]

introduced *CodeShovel*, a state-of-the-art tool capable of recovering method histories despite various transformations. On a manually annotated oracle, *CodeShovel* outperformed both research tools (e.g., *FinerGit*) and industry tools (e.g., *IntelliJ*, *GitLog*). However, Jodavi *et al.* [60] found that *CodeShovel* struggles when methods undergo significant body changes, such as during moves, often failing to track them correctly or misattributing their origin.

By leveraging the RefactoringMiner [143], the authors also noted inaccuracies in the original *CodeShovel* oracle and revised it based on their interpretation to create a more reliable benchmark. In general, they identified inaccuracies in 27 method histories due to method extraction refactorings. The authors updated the *CodeShovel* oracle based on RefactoringMiner’s suggestions, but this approach has two key issues: (i) the accuracy of the remaining 173 methods is unknown, and (ii) RefactoringMiner itself has low recall (54.2) for the extract and move method [143], raising doubts about the updates’ reliability. The authors also developed *CodeTracker*, which employs RefactoringMiner’s advanced entity matching algorithm [143]. When evaluated with the new oracle, *CodeTracker* showed significant improvements in precision and recall over *CodeShovel*, but its reliance on RefactoringMiner introduced a performance bottleneck, making it significantly slower.

Motivation: Like its predecessor, the new *CodeTracker* oracle remains subject to internal validity threats, as it relies on the RefactoringMiner and on researchers’ interpretations. A more systematic oracle construction process is needed for the reliable evaluation of method history tools. Additionally, there is a need for a new algorithm that maintains high accuracy without sacrificing runtime performance, unlike *CodeTracker*.

3.3 Developing the New Oracles

To evaluate method-tracking tools, it is essential to have reliable ground truths against which the correctness of the tools can be measured. An oracle created in a purely manual manner remains vulnerable to internal validity threats due to its substantial dependence on human

judgment. Manual efforts are prone to inconsistencies, oversight, and bias, particularly when dealing with complex software evolution patterns. Constructing such an oracle is also highly impractical for large-scale real-world codebases, which often consist of thousands of files and methods. During software evolution, methods may undergo renaming, move across files, or be refactored in tandem with other code changes, including modifications to their signatures and bodies. These transformations may occur within a single commit or span across multiple commits, making manual tracking extremely error-prone and difficult to scale.

Therefore, a hybrid approach was adopted that combined automated analysis with human verification. Method histories were collected with multiple state-of-the-art tools, each with distinct strengths and weaknesses, and each reported change was manually validated to ensure high precision and recall. This union-based strategy was inspired by an observation in the original *CodeShovel* research: although *CodeShovel* outperformed all other tools, it occasionally missed method change commits that were correctly identified by those other tools.

To find potential change commits that were not captured by any of the tools, the first two authors used the `git log` command to trace changed files (not methods) containing the methods of interest, manually verifying whether any were actual change commits. These two authors have 8+ years of industry experience in software development and **Git-based** project management. This process uncovered 43 additional commits. While this quick manual process may not reveal all missing commits, finding only 43 out of 9,005 total suggests that the union set already had high recall.

Each method’s history was then independently reviewed by these two authors. Inter-rater agreement was evaluated using Cohen’s Kappa [99], which yielded a value of 0.97, indicating an almost perfect [100] level of agreement between the two authors. Out of 9,005 commits in the union set, only 66 had disagreements, which arose from: (i) introduction of new overloaded versions with the same name, and (ii) different tools identifying different methods due to varying body-similarity thresholds. These disagreements were resolved through Zoom

Collect Method History

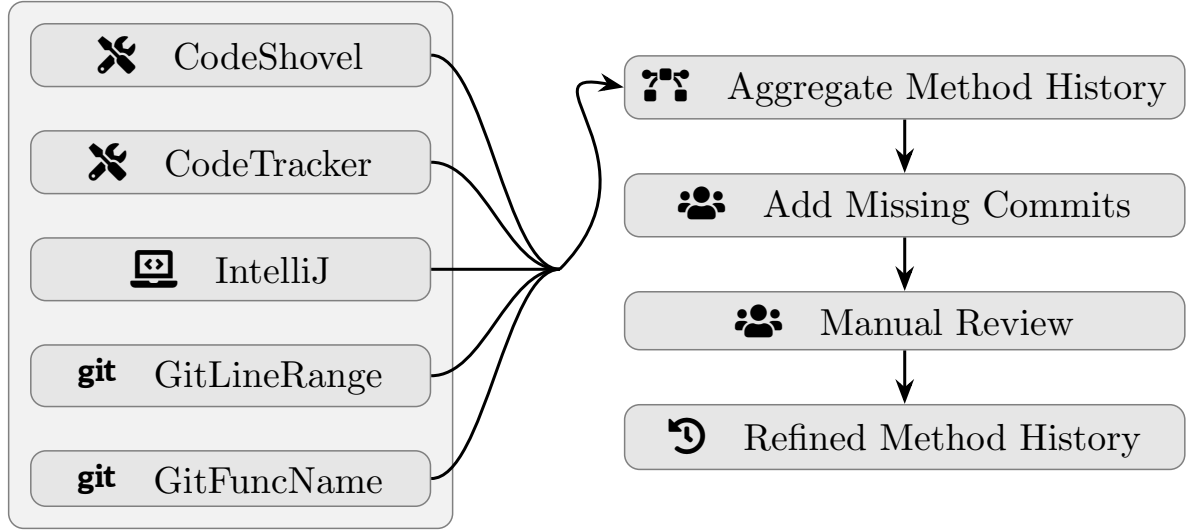


Figure 3.1: The process used to generate accurate oracles. The union of the change commits reported by all tools was first created, missing commits were added, and false positives were then removed.

discussions by examining function call locations and contextual information in the Java file. Figure 3.1 summarizes the oracle construction process. The next sections describe the two oracles developed here.

3.3.1 Updating the CodeShovel Oracle

Using the five tools (*CodeShovel*, *CodeTracker*, *IntelliJ*, *GitLineRange*, and *GitFuncName*), method change histories were collected for the 200 methods used in the original *CodeShovel* study. For *CodeShovel* and *CodeTracker*, the tools were used as library dependencies, as they are publicly available on GitHub. For *IntelliJ*, each project was manually imported, the specific commits were checked out, the target files and methods were located, and the **Show History for Selection** option was used to generate the method change histories. For the Git-based tools, shell commands namely `git log <commit> --no-merges -L :<funcname>:<file>` and `git log <commit> --no-merges -L <start>,<end>:<file>` were executed to collect method histories. Finally, the method change histories from all five tools

were aggregated into a unified commit history, producing a union of the commit histories of all tools. The missing commits mentioned earlier were also added. The commit histories were then sorted in reverse chronological order by their commit timestamp. Next, each method change commit was manually inspected to verify its logical validity by comparing it with its parent commit. This process is repeated iteratively for each subsequent commit until all commits have been thoroughly examined. At the end of this process, a complete and accurate method history oracle was obtained for the 200 methods. These 200 methods are referred to as the corrected *CodeShovel* oracle.

3.3.2 Constructing the HistoryFinder Oracle

The *HistoryFinder* oracle was created from 20 additional open-source Java projects selected from **GitHub**. To ensure high-quality projects [97], popular projects were selected based on a high number of stars and forks. Each project was also required to have at least 2,000 commits so that enough method change commits could be found for the selected methods from these projects. Table 3.1 presents the list of selected projects along with their number of commits, Java files, methods, and their respective star and fork counts. From each project, 10 methods were randomly selected, recording their file names, method names, and start and end line numbers, resulting in a total of 200 methods. The same approach described in section 3.3.1 was then followed to generate the change history oracle for this new set of 200 methods.

3.4 Developing the HistoryFinder Tool and Algorithm

The first, third, and fourth authors collaboratively designed the *HistoryFinder* algorithm through multiple Zoom meetings. The first author has 8+ years of industry experience; the third has 14+ years in industry and 3+ in research; and the last has 1 year in industry and 15 years in research. During these sessions, they analyzed the strengths and limitations

Table 3.1: Overview of the 20 Java repositories used for the *HistoryFinder* oracle, including commit counts, source files, methods, and popularity metrics (stars and forks).

Repository	# Commits	# Files	# Methods	# Stars	# Forks
auto	2,570	317	4,211	10,441	1,202
cassandra	31,705	4,926	76,475	8,913	3,641
dagger	5,592	1,436	8,043	17,467	2,019
dubbo	9,428	3,713	28,365	40,581	26,442
fastjson	7,216	2,997	19,239	25,761	6,498
glide	3,395	724	8,804	34,718	6,128
guice	2,444	615	6,981	12,521	1,673
Hystrix	2,354	415	6,436	24,160	4,708
j2objc	6,314	5,119	106,062	5,998	972
jenkins	37,342	1,779	21,660	23,363	8,834
jmeter	32,202	1,526	16,138	8,468	2,116
kafka	20,356	4,632	62,991	29,077	14,030
MPAndroidChart	2,074	223	2,447	37,694	9,027
netty	19,625	1,973	22,902	33,578	15,974
pulsar	20,639	4,153	44,183	14,323	3,601
rocketmq	8,918	1,581	17,155	21,361	11,727
RxJava	7,698	1,884	33,994	47,951	7,602
shardingsphere	43,769	7,765	34,921	20,035	6,775
tomcat	77,289	2,705	42,021	7,619	5,062
zxing	3,816	499	3,293	32,923	9,365

of two existing approaches, *CodeShovel* and *CodeTracker*, which guided the design of their new algorithm. Their analysis revealed that while *CodeTracker* is generally more accurate than *CodeShovel*, it suffers from significantly slower runtime performance. This is primarily due to its dependence on RefactorMiner, whereas *CodeShovel* uses a simpler, faster string-matching algorithm. To balance accuracy and efficiency, *HistoryFinder* adopts *CodeShovel*'s lightweight string-matching approach but improves upon it with a more precise method-tracking strategy. The main algorithmic differences are as follows.

- In Phase 2 of the *CodeShovel* algorithm, the first method with body similarity ≥ 0.75 is selected, without checking for a matching signature. This can lead to tracking an extracted method instead of the original. In contrast, *HistoryFinder* first checks for a

matching signature and resorts to body similarity only if none is found.

- When the method is not found in the same file, *CodeShovel* performs two phases: first, it looks for methods with the same signature (even with low body similarity), then for methods with high body similarity. During the training phase, a method in a different file with the same signature but low similarity was observed to not necessarily be the same method. Merging these phases—directly searching for highly similar methods without checking the signature—improves precision.
- Additionally, in Phase 4, *CodeShovel* compares all methods across modified files, which is not only time-consuming but can also lead to incorrect matches, especially in the case of method overloading. This issue was addressed by filtering out methods that were unchanged in both commits, as such methods cannot be the ones of interest. Also, *CodeShovel* returns the first score exceeding the threshold, which is problematic when a more similar method exists. To alleviate this issue, *HistoryFinder* returns the best matching method passing the similarity threshold.

HistoryFinder also improved some of the technical weaknesses that *CodeShovel* has. For example, during development, if code is merged without rebasing, conflicts can arise when a method is modified concurrently in different branches. The resolution of these conflicts may introduce changes to the method’s history that must be considered. *CodeShovel*, however, excludes merge commits and ignores changes introduced during merges. In this approach, merge commits are included, and the method’s state is compared with each parent commit, allowing accurate detection of any changes introduced from either parent branch. Moreover, *CodeShovel* does not account for formatting changes, annotation, or JavaDoc updates, whereas *HistoryFinder* does. Another limitation of *CodeShovel* is its case-insensitive string-matching algorithm. This approach can be problematic, especially when comparing quoted string literals within a method body, as it may overlook subtle but important differences that could influence the method’s behavior.

3.4.1 Overview of HistoryFinder

The *HistoryFinder* tool needs several key inputs to trace the history of a method. These include the `Git` repository URL or local directory, a specific commit hash (SHA), the file path of the method (including the file name), the name of the method, and the line number where the method declaration begins. The output is a sequential list of commit hashes, each accompanied by relevant metadata. This metadata provides granularity of changes, such as changes to the method signature, body, and annotations, and identifies the contributor responsible for each modification. When executed via the command line, the output is saved as a JSON file.

The implementation utilizes the `JGit` library²—a Java-based `Git` implementation—for repository cloning, commit traversal, and fine-grained file operations, including detection of added, deleted, moved, and modified files across revisions, and employs `JavaParser`³ to generate an Abstract Syntax Tree (AST). Once the AST is constructed, it allows traversal through the list of methods in the file and extraction of specific components, such as the method signature, body, annotation, and `JavaDoc`. For method comparison, the implementation uses the Jaro-Winkler edit distance [144] score similarity, as in *CodeShovel*, between method signatures and bodies. Although `SimHash` [145] was also tested for similarity matching, Jaro-Winkler consistently provides better recall than `SimHash`.

When the commit history is interpreted as a hierarchical graph structure, all commits leading up to a specific point in a `Git` project form a Directed Acyclic Graph (DAG) [146], representing the project’s complete commit history to that point. To identify all commits that modified a method up to a given version (e.g., the starting commit), the DAG is traversed backward—from the target commit to its ancestors—until the commit where the method was originally introduced is reached. During this traversal, if a commit has a single parent, the method in the current commit is compared with its counterpart in the parent

²<https://github.com/eclipse-jgit/jgit> last accessed: Jul 01, 2025

³<https://github.com/javaparser/javaparser> last accessed: Jul 01, 2025

commit to detect changes. In the case of merge commits with multiple parents, the method may exist in any of the parent commits. Given that a project may contain thousands of commits (as shown in Table 3.1), traversing the entire DAG can be time-consuming. To optimize this process, *HistoryFinder* leverages the `git log <commit> <file>` command, which filters the commit history to only those that modified the file containing the target method. This significantly reduces the search space, allowing traversal of a lightweight, file-specific DAG to detect method changes efficiently. However, if the method has been moved to a different file or the file itself has been renamed, a new DAG must be reconstructed from that point onward using the new file path.

3.4.2 The HistoryFinder Algorithm

Algorithm 1 outlines how the file-specific DAG is constructed and traversed to detect the change history of a method. The procedure *HistoryFinder* is initially invoked with the starting commit, file path, method name, and its starting line number. It may be invoked recursively if the method is detected to have moved to a different file or if the file itself has been renamed. To build the DAG, the utility procedure *GITLOG* is first called (line 2), which internally executes the `git log <commit> <file>` command. This retrieves only the commits that have modified the target file containing the method. Using these filtered commits, a directed acyclic graph G is constructed, where each node represents a commit that changed the file. In the DAG, an edge is drawn from node u to node v if the file is modified in commit u , followed by in commit v , establishing a direct connection between them. Once the DAG is constructed, it is traversed using the Breadth-First Search (BFS) algorithm [147] to systematically explore all relevant commits.

Queue Q keeps track of all discovered commit nodes that need to be explored. To unambiguously locate the target method in a commit, each commit node is represented by a structure containing the commit hash, file path, method name, and line number, created by invoking `Create` (line 3 for the initial node) procedure. The queue Q is initialized with

the starting node (line 4), which is also the only entry in the set `visited` to mark it as explored. Whenever a change to the method is detected, it is recorded in the history list H , which is initially empty (line 6). The `while` loop iterates through all discovered nodes. In each iteration, an unexplored commit node u is dequeued from the head of Q (line 8). The method already found in commit node u is located by calling the procedure `Find` (line 9), which takes the commit hash, file, method name, and line number as parameters, accessed using dot notation. Next, the parent node p of u is retrieved (line 10), where the method might be found. From lines 11 to 18, the algorithm checks whether the method exists in the direct parent commit p , accounting for possible modifications.

The algorithm first tries to locate the method by matching its signature (line 11), including the method name and parameters, but excluding the return type. If the signature match fails, it attempts to locate the method by the textual similarity of its body (line 14), considering a match valid if the similarity is at least 70%. If the method still cannot be found, the search expands to other modified files (line 16), raising the similarity threshold to 75% in this case. If the method is found in a different file—due to a method move or a file rename/move—the new file path is recorded in `altFile`. When the method is located through any of the above mechanisms (line 19), the algorithm checks whether the methods (from the current and parent commits) are identical, including their signatures, JavaDocs, annotations, or whether the file has been renamed or moved (in which case `altFile` should not be `NIL`) (line 20). If any change is detected—either in the method or in the file name or path—this change is recorded by invoking the `Add` procedure (line 21).

If the method is found in another file, a new DAG must be reconstructed for that file. To do so, the `HistoryFinder` procedure is recursively called (line 24), returning a sub-history that is appended to H (line 25). If the method is instead found within the same file (line 27), all immediate ancestor commit nodes are processed. In this case, all adjacent ancestor commit nodes of u that have not yet been visited are added to the visited set (line 30) and enqueued (line 33). The method name (line 31) and line number (line 32) are also updated

Algorithm 1: HistoryFinder Algorithm

```
1 procedure HistoryFinder(hash, file, name, line):  
   Input: hash: Start commit hash  
           file: File containing the method  
           name: Method name  
           line: Start line of the method  
   Output: Change history of the input method  
2    $G \leftarrow \text{GITLOG}(\textit{hash}, \textit{file});$  // Build file-level DAG  
3    $s \leftarrow \text{CREATE}(\textit{hash}, \textit{file}, \textit{name}, \textit{line});$   
4    $Q \leftarrow \{s\};$  // Initialize queue with start node  
5    $\textit{visited} \leftarrow \{s\};$  // Track visited nodes  
6    $H \leftarrow \emptyset;$  // Initialize with empty method history  
7   while  $Q \neq \emptyset$  do  
8      $u \leftarrow \text{DEQUEUE}(Q);$   
9      $um \leftarrow \text{FIND}(u.\textit{hash}, u.\textit{file}, u.\textit{name}, u.\textit{line});$   
10     $p \leftarrow \text{PARENT}(G, u);$  // Direct parent commit  
    // Search method by signature, body in original file, then body in other  
    // files  
11     $m \leftarrow \text{SIGNATUREMATCH}(p, \textit{file}, um);$   
12     $\textit{altFile} \leftarrow \text{NIL};$  // Track method/file move  
13    if  $m == \text{NIL}$  then  
14       $m \leftarrow \text{BODYMATCH}(p, \textit{file}, um, 0.70);$   
15      if  $m == \text{NIL}$  then  
16         $m, \textit{altFile} \leftarrow \text{ALTFILEMATCH}(u.\textit{hash}, p, um);$   
17      end if  
18    end if  
    // Method found in original/other file  
19    if  $m \neq \text{NIL}$  then  
20      if  $um.\textit{text} \neq m.\textit{text}$  or  $\textit{altFile} \neq \text{NIL}$  then  
21         $\text{ADD}(H, u);$  // Record method change  
22      end if  
      // Continue tracking in the new file  
23      if  $\textit{altFile} \neq \text{NIL}$  then  
24         $sH \leftarrow \text{HISTORYFINDER}(m.\textit{hash}, m.\textit{file}, m.\textit{name}, m.\textit{line});$   
25         $\text{ADDALL}(H, sH);$  // Append sub-history  
26      end if  
      // Enqueue unvisited ancestor commits  
27      if  $m \neq \text{NIL}$  and  $\textit{altFile} == \text{NIL}$  then  
28        foreach  $v \in G.\textit{Adj}[u]$  do  
29          if  $v \notin \textit{visited}$  then  
30             $\text{ADD}(\textit{visited}, v);$   
31             $v.\textit{name} \leftarrow m.\textit{name};$   
32             $v.\textit{line} \leftarrow m.\textit{line};$   
33             $\text{ENQUEUE}(Q, v);$   
34          end if  
35        end foreach  
36      end if  
37    end if  
38  end while  
39  return  $H;$   
40 end procedure
```

Algorithm 2: Utility Procedures for Method Matching

```
1 procedure SignatureMatch(hash, file, tm):  
    Input: hash: Commit hash  
           file: File containing methods  
           tm: Target method  
    Output: Matching method or NIL if not found  
2    methods  $\leftarrow$  PARSEMETHOD(hash, file);  
3    foreach m  $\in$  methods do  
4        if m.signature == tm.signature then  
5            return m ;                               // Return matching method  
6        end if  
7    end foreach  
8    return NIL ;                                       // No matching method found  
9 end procedure  
  
10 procedure BodyMatch(hash, file, tm, threshold):  
    Input: hash: Commit hash  
           file: File containing methods  
           tm: Target method  
           threshold: Similarity threshold for matching  
    Output: bm: Best matching method meeting threshold  
           bs: Similarity score of the best match  
11    methods  $\leftarrow$  PARSEMETHOD(hash, file);  
12    bm  $\leftarrow$  NIL ;                               // Initialize best matching method  
13    bs  $\leftarrow$  0 ;                                   // Initialize highest similarity score  
14    foreach m  $\in$  methods do  
15        score  $\leftarrow$  SIMILARITY(m.text, tm.text);  
16        if score  $\geq$  threshold and score  $>$  bs then  
17            bm  $\leftarrow$  m ;                               // Update best matching method  
18            bs  $\leftarrow$  score ;                           // Update highest similarity  
19        end if  
20    end foreach  
21    return bm, bs;  
22 end procedure  
  
23 procedure AltFileMatch(hash, parentHash, tm):  
    Input: hash: Commit hash  
           parentHash: Parent commit hash  
           tm: Target method  
    Output: bm: Best matching method  
           altFile: File containing the best matching  
24    changeFiles  $\leftarrow$  FINDCHANGEFILES(hash, parentHash);  
25    bm  $\leftarrow$  NIL ;                               // Initialize best matching method  
26    bs  $\leftarrow$  0 ;                                   // Initialize highest similarity score  
27    altFile  $\leftarrow$  NIL ;                             // Best method matching file  
28    foreach file  $\in$  changeFiles do  
29        m, s  $\leftarrow$  BODYMATCH(parentHash, file, tm, 0.75);  
30        if s  $>$  bs then  
31            bm  $\leftarrow$  m;  
32            bs  $\leftarrow$  s;  
33            altFile  $\leftarrow$  file;  
34        end if  
35    end foreach  
36    return bm, altFile;  
37 end procedure
```

to reflect the newly found version, so that it can be retrieved accurately when this node is dequeued later. Once all nodes have been processed, the resulting sequence of commit nodes captures the method’s change history in H . Additionally, the furthest node dequeued from the queue Q is added as the introduction commit, since this is the earliest known instance of the method.

Algorithm 2 defines the utility procedures used to compare and match methods based on textual similarity. The procedure **SignatureMatch** attempts to locate the target method within a Java file at a specific commit. Line 2 parses all methods in the file using **JavaParser**, and lines 3 to 7 iterate over these methods, comparing each one’s signature with the target method. If a matching signature is found, the corresponding method is returned (line 5); otherwise, **NIL** is returned (line 8). The next procedure, **BodyMatch** (line 10), identifies a method based on the highest body similarity with the target method, provided it exceeds a specified similarity threshold. A threshold of 70% is used when comparing methods within the same file, and 75% when comparing across different files. Line 11 parses all Java methods in the file provided as input. The variable **bm** (line 12) tracks the best matching method, while **bs** (line 13) stores the corresponding similarity score. Line 14 iterates over all parsed methods, and the **Similarity** procedure (line 15) computes the similarity between the current method and the target method using the Jaro-Winkler algorithm [144]. If the computed similarity meets the threshold and is greater than the current best score **bs**, both the score and the best matching method are updated (lines 16–19). Finally, the best matching method along with its similarity score is returned (line 21).

The procedure **AltFileMatch** attempts to locate the target method in all files that were modified in a given commit. It begins by listing all modified files using the **FindChangedFiles** procedure (line 24). Lines 25 to 27 initialize variables to track the best matching method, its similarity score, and the corresponding file path. Each modified file is then iterated (line 28), and the best matching method within each file is identified (line 29). If a method with a higher similarity score is found, the method information is updated accordingly (lines 30–34).

Finally, the method with the highest similarity score and the new file path are returned (line 36). Similar to the *CodeShovel* algorithm [45], the first 100 methods from the corrected *CodeShovel* oracle were used to train the algorithm and determine the optimal thresholds for body similarity within the original and other files.

3.5 Analysis and Results

This section compares the corrected *CodeShovel* oracle with the original *CodeShovel* and *CodeTracker* oracles. It then evaluates the accuracy and runtime efficiency of all method change history generation tools using the corrected *CodeShovel* and the newly developed *HistoryFinder* oracles. In particular, it answers the following three research questions.

- **RQ1:** To what extent does the corrected *CodeShovel* oracle differ from the original oracles?
- **RQ2:** Which history generation tool exhibits the highest accuracy when evaluated with the two new oracles?
- **RQ3:** Which history generation tool exhibits the best runtime performance?

3.5.1 RQ1: Difference with the corrected CodeShovel oracle

After correcting the 200 method histories in both the *CodeShovel* and *CodeTracker* oracles, the new systematically constructed oracle—the *corrected CodeShovel oracle*—was compared with the original *CodeShovel* and *CodeTracker* oracles, revealing significant updates. Table 3.2 provides a detailed breakdown of the changes. It shows how many methods were *changed*, commits were *added*, and *removed*. A method is considered *changed* if the new oracle has a different set of commits than the original. A commit is considered *added* if it was not present in the original oracles, and *removed* if it was inaccurately included. For example, the last column shows that the *CodeTracker* oracle has fewer inaccurate commits

Table 3.2: Comparison between the corrected and original oracles. As the corrected oracle includes changes related to annotations, JavaDocs, and formatting, the differences are shown both including (*Incl.*) and excluding (*Excl.*) those changes.

Oracle	# Methods		# Commits		
	# Changed		# Added		# Removed
	Incl.	Excl.	Incl.	Excl.	
CodeShovel	142	40	429	59	186
CodeTracker	133	43	445	134	67

(67) than the original *CodeShovel* oracle. In contrast, it missed many commits that should have been recorded as change commits for those 200 methods, as shown in the *# Added* column.

Compared to the original *CodeShovel* oracle, the histories of 40 methods were *modified*, with 59 commits *added* and 186 *removed*. These changes exclude annotations, JavaDocs, and formatting—omitted by the original *CodeShovel* and *CodeTracker* oracles, though *CodeTracker* did include annotations. When these types of changes were included, 142 method histories were affected, and 429 commits were *added*. Significant changes were also observed when compared with the *CodeTracker* oracle. These changes suggest that the previous evaluations of method tracing tools (e.g., *CodeShovel* and *CodeTracker*) may not reflect the true precision and recall of these tools in tracing a method’s change history.

To illustrate inaccuracies in both the *CodeShovel* and *CodeTracker* oracles, several cases are highlighted. One representative example is the `fireErrors` method from the `checkstyle` project. Both the *CodeShovel*⁴ and *CodeTracker*⁵ oracles record this method’s history beginning from commit 119fd4, where the method resided in `src/main/java/com/puppycrawl/tools/checkstyle/Checker.java`. However, both oracles incorrectly identified commit 0e3fe as the method’s introduction, whereas it was actually introduced earlier in commit 0fd69 (the method name was `displayErrors` and was later renamed to `fireErrors`).

⁴<https://github.com/SQMLab/HistoryFinder/blob/main/oracle/codeshovel-oracle/checkstyle-Checker-fireErrors.json>

⁵<https://github.com/SQMLab/HistoryFinder/blob/main/oracle/codetracker-oracle/checkstyle-Checker-fireErrors.json>

This error was recently corrected in the *CodeTracker* oracle; however, the evaluation was conducted using the original published version, where the issue was present.

In another case (`createPattern` method) in the *CodeShovel* oracle⁶, while saving the history of an overloaded method, the history failed to stop at the point where the overloaded method was introduced and inaccurately switched to tracing the original of the overloaded method, resulting in incorrect method changes in the oracle. Another challenge arises when handling merge commits. While the *CodeShovel* oracle completely ignored changes that occurred in merge commits, the *CodeTracker* oracle occasionally added incorrect method changes. An example of this issue occurs in the `runChild` method of the `junit4` project. Although the diff⁷ appears to show a method change, it is incorrect—the commit has two parents,⁸ and the comparison was made against the wrong parent.⁹ In reality, the code was introduced from the other branch¹⁰ without any modification.

RQ1 summary: Both the *CodeShovel* and *CodeTracker* oracles contain inaccuracies in their constructed method histories. The updated, more accurate oracle can serve as a valuable resource for effectively evaluating both existing and future method-level change history generation tools.

3.5.2 RQ2: Accuracy of the change history generation tools

The accuracy of six method change history generation tools, including the newly proposed *HistoryFinder*, is evaluated next. This evaluation uses the corrected *CodeShovel* oracle (200

⁶<https://github.com/checkstyle/checkstyle/compare/d2551035044a845fd1b3e345f2470875a43a8991...f2c6263e151e8a7300755b36fbb41511c0a0cca1#diff-1b66e778c19cac5ac2beaec4d6a74c8c37a5fd83d39db4bef8c1b60b9ec68c0e> last accessed: July 3, 2026

⁷<https://github.com/junit-team/junit4/compare/8ff0b44e3fb0c1c56a1dc6290c3b6828a5a8f9bf...bed58a573c373d57d64fa369f58b2a8f0dbec3d7#diff-cd3bb9170f8d6404c5dc5bb66f7b647435854d48fe357e0bd998e3011a575929> last accessed: July 3, 2026

⁸<https://github.com/junit-team/junit4/blob/a49240ade1974b948b20cf2c45d9129f04122735/src/main/java/org/junit/runners/BlockJUnit4ClassRunner.java#L64> last accessed: July 3, 2026

⁹<https://github.com/junit-team/junit4/blob/708ed373c02b467422890d5163fff46d1f32ab01/src/main/java/org/junit/runners/BlockJUnit4ClassRunner.java#L66> last accessed: July 3, 2026

¹⁰<https://github.com/junit-team/junit4/blob/9d8bb069f68e2194db742981972c8930381b62c2/src/main/java/org/junit/runners/BlockJUnit4ClassRunner.java#L65> last accessed: July 3, 2026

methods) and the new *HistoryFinder* oracle (another 200 methods). The *CodeShovel* oracle is divided into two parts: *CodeShovel Training* and *Testing* oracles. This distinction is important, as both the *CodeShovel* and *HistoryFinder* algorithms were trained and tuned using the training portion of the *CodeShovel* oracle. The three research-based tools were run as Java libraries to extract method histories and compute accuracy metrics. For Git-based tools, the command `git log <commit> --no-merges -L :<funcname>:<file>` was used to generate method histories for *GitFuncName*, which tracks changes by function name, and `git log <commit> --no-merges -L <start>,<end>:<file>` was used for *GitLineRange*, which tracks changes by line range. In contrast, for *IntelliJ*, IntelliJ IDEA 2024.3 Community Edition was used to manually generate method histories (**Right-click** → **Git** → **Show History for Selection**), as no automated approach exists for running IntelliJ’s method tracking feature.

Precision, recall, and F1 score were calculated for all the tools against the three oracles. Since tools differ in the types of changes they detect (e.g., *CodeShovel* ignores annotations, *CodeTracker* ignores formatting), the expected commit sets were tailored to match each tool’s capabilities, ensuring a fair and meaningful comparison. The tools’ accuracy is evaluated at two granularities: *commit-level* and *method-level*. At the *commit-level*, precision, recall, and F1 scores are computed by aggregating true positives (TP), false positives (FP), and false negatives (FN) across each oracle, based on the total expected commits. At the *method-level*, metrics are computed per method and averaged across each oracle.

Table 3.3 shows that all research-based tools—*CodeShovel*, *CodeTracker*, and *HistoryFinder*—significantly outperform *IntelliJ* and **Git-based** tools. Overall, *HistoryFinder* achieves the highest F1 score. In terms of precision, *HistoryFinder* and *CodeTracker* slightly outperform *CodeShovel*. For recall, *CodeShovel* performs best on its own training oracle; however, on the other two oracles, *HistoryFinder* achieves the highest recall. *CodeShovel*’s notably lower recall on the *HistoryFinder* oracle is primarily due to its failure to generate the complete history for several methods. For example, it was unable to produce complete

Table 3.3: Precision, recall, and F1 score for method history tracking tools across three oracles. TP (True Positives) indicates correctly identified change commits; FP (False Positives) refers to incorrectly included commits; FN (False Negatives) denotes missed commits that should have been identified.

Oracle	Tool	<i>Commit-Level</i>						<i>Method-Level</i>		
		TP	FP	FN	Precision	Recall	F1	Precision	Recall	F1
CodeShovel Training	CodeShovel	2813	158	28	94.68	99.01	96.80	94.79	99.31	95.86
	CodeTracker	2804	32	76	98.87	97.36	98.11	98.99	96.98	97.52
	HistoryFinder	3047	6	67	99.80	97.85	98.82	99.68	98.15	98.53
	IntelliJ	2146	402	919	84.22	70.02	76.47	85.16	75.08	73.40
	GitLineRange	2197	1135	787	65.94	73.63	69.57	77.94	76.32	68.29
	GitFuncName	1749	1166	1261	60.00	58.11	59.04	72.93	62.03	54.88
CodeShovel Testing	CodeShovel	812	25	38	97.01	95.53	96.27	97.40	95.70	95.54
	CodeTracker	811	35	63	95.86	92.79	94.30	97.59	94.36	94.76
	HistoryFinder	938	10	7	98.95	99.26	99.10	99.15	99.24	99.15
	IntelliJ	685	239	249	74.13	73.34	73.74	85.14	72.58	73.78
	GitLineRange	609	288	279	67.89	68.58	68.24	87.48	70.02	72.31
	GitFuncName	579	393	331	59.57	63.63	61.53	77.85	65.34	65.35
HistoryFinder	CodeShovel	2095	147	394	93.44	84.17	88.56	95.68	86.45	85.82
	CodeTracker	2354	88	168	96.40	93.34	94.84	94.43	91.24	94.14
	HistoryFinder	2835	106	48	96.40	98.34	97.36	96.49	96.81	97.00
	IntelliJ	2185	1530	637	58.82	77.43	66.85	75.99	81.38	75.80
	GitLineRange	2032	2112	616	49.03	76.74	59.84	77.64	82.98	75.50
	GitFuncName	1858	2638	835	41.33	68.99	51.69	65.08	72.67	59.50

histories for all 10 methods from the Cassandra project. Additionally, it failed entirely for two methods in the Fastjson project due to `JavaParser` errors that prevented successful file parsing. Overall, across all three oracles, *HistoryFinder* demonstrates the most consistent and superior performance. Results at the *method-level* scores closely correlate with *commit-level* scores, with minor variations. For instance, *CodeShovel*'s precision against the *HistoryFinder* oracle is approximately 2% higher at the *method-level* compared to the *commit-level*. The Wilcoxon rank-sum test [148] also shows that the observed accuracy gain with *HistoryFinder* is statistically significant ($P \leq 0.05$).

RQ2 summary: The *HistoryFinder* algorithm consistently achieves the highest precision and F1 score in change history detection. Between *CodeShovel* and *CodeTracker*, performance varies depending on the scenario. Notably, *CodeShovel*'s recall drops significantly on the entirely new oracle of 200 methods, mainly because it fails to generate the full or any history for a subset of methods. Given these findings, researchers or practitioners who prioritize accuracy should consider *HistoryFinder* the most reliable option.

3.5.3 RQ3: Runtime performance of the change history generation tools

The tools were executed in a controlled environment on a machine equipped with *32GB of RAM, an Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz with 12 cores, running Ubuntu 22.04 LTS and Java 21 x86*. Execution time for *IntelliJ* was not recorded since method histories were generated manually from *IntelliJ IDEA*, making it infeasible to measure the time. Prior to recording execution time, all repositories were cloned to a local persistent storage. Initially, the JAR dependencies of the research-based tools were integrated. However, caching of parsed Java files across multiple method history runs—particularly in *CodeShovel*—was observed to significantly reduce the runtime. While this optimization benefits batch processing, it does not yield deterministic or representative runtimes for tracing individual methods. To ensure a fair comparison, each tool was instead executed independently for each method history using the `java -jar` command. This approach ensured that each method's history was processed in isolation within a fresh environment, by launching a separate *JAR* process for each run and terminating it upon completion. Please note that due to the deterministic nature of the evaluated tools, the difference in execution times in multiple runs for a given tool is negligible.

Figure 3.2 shows the distribution of execution times for all five tools across the three or-

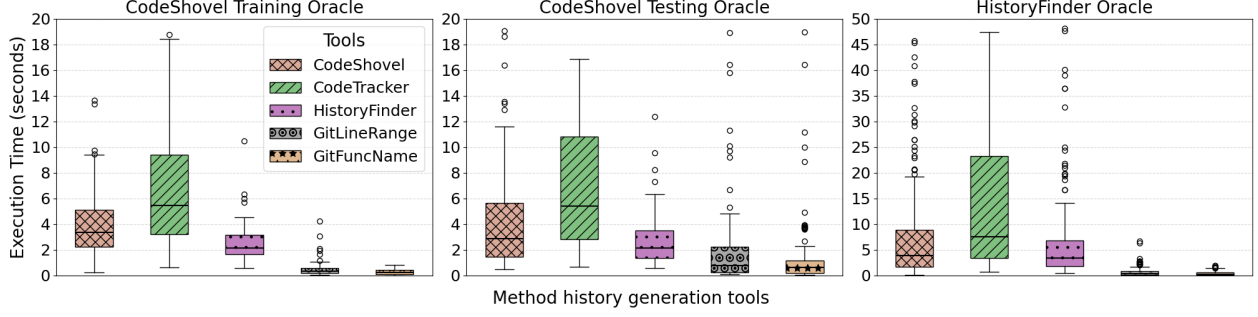


Figure 3.2: Comparison of execution times across three oracles for five tools. To enhance readability, the y-axis is limited to 20 seconds for the first two subplots (*CodeShovel* oracles) and 50 seconds for the third subplot (*HistoryFinder* oracle).

acles, while Table 3.4 presents the mean, median, minimum, and maximum execution times. Excluding outliers, *HistoryFinder* demonstrates the lowest execution times among the three research-based tools. In general, *CodeTracker* is the slowest in constructing method change history. Across all tools and oracles, the minimum execution time consistently remained below one second. However, for the research-based tools, although the mean and median execution times are slightly higher in the *HistoryFinder* oracle compared to the *CodeShovel* oracle, the difference in maximum execution time between these two oracles is unexpectedly large.

To understand the factors contributing to the increased runtime in the *HistoryFinder* oracles, cases with unusually high execution times were manually examined. Prolonged runtimes were often linked to a high frequency of changes in the file containing the target method, regardless of whether the method itself was modified. In such cases, parsing a high number of Java files using the **JavaParser** was particularly time-consuming. Profiling of the tools revealed that, for methods with high execution times, over 80% of the total runtime was typically spent on parsing alone, thereby significantly increasing the computational overhead. Repositories like Kafka, Cassandra, Jenkins, and Pulsar frequently exhibited these patterns, and these types of projects are more prevalent in the *HistoryFinder* oracle compared to the *CodeShovel* oracle.

CodeTracker exhibited significantly high execution times when a method or its enclos-

Table 3.4: Execution times (in seconds) for the five tools. *IntelliJ* is excluded because of the need for manual execution.

Oracle	Tool	Mean	Median	Min	Max
CodeShovel Training	CodeShovel	3.93	3.38	0.22	13.63
	CodeTracker	8.71	5.45	0.64	76.80
	HistoryFinder	2.51	2.15	0.59	10.48
	GitLineRange	0.50	0.35	0.07	4.23
	GitFuncName	0.26	0.24	0.02	0.83
CodeShovel Testing	CodeShovel	4.55	2.86	0.49	26.08
	CodeTracker	9.29	5.42	0.68	63.34
	HistoryFinder	2.85	2.15	0.57	20.78
	GitLineRange	2.13	0.80	0.11	18.93
	GitFuncName	1.57	0.61	0.07	18.94
HistoryFinder	CodeShovel	7.74	3.88	0.15	74.49
	CodeTracker	30.41	7.56	0.76	693.66
	HistoryFinder	7.32	3.47	0.49	106.64
	GitLineRange	0.69	0.38	0.04	6.66
	GitFuncName	0.41	0.21	0.03	1.96

ing file was moved in a commit that also modified many other files—a pattern observed in repositories such as Netty and ShardingSphere. In contrast, *CodeShovel*’s maximum execution time on the *HistoryFinder* oracle was much lower than that of *CodeTracker* and *HistoryFinder*. However, this lower runtime is primarily due to *CodeShovel* failing to generate complete histories for many complex methods with extensive change histories—cases where both *CodeTracker* and *HistoryFinder* succeeded.

The **Git-based** tools, in contrast, did not exhibit high execution times in the *HistoryFinder* oracle. Their maximum execution times were also significantly lower than those of the research-based tools. This lower runtime is primarily due to two factors: (i) **Git-based** tools operate using `git log` without relying on language-specific parsers. Instead, they track line-level changes, making their execution comparatively lightweight; and (ii) similar to *CodeShovel*, these tools often failed to recover the complete change histories of complex methods. As a result, their efficiency comes at the cost of significantly lower precision and

recall compared to their research-based counterparts, as also reflected in Table 3.3.

RQ3 summary: `Git`-based tools exhibit the lowest execution times, but this efficiency comes at the cost of significantly reduced accuracy. Among research-based tools, *HistoryFinder* strikes the best balance of speed and accuracy, while *CodeTracker*, the second most accurate, shows the highest and most variable runtimes. Overall, *HistoryFinder* is the most reliable choice for those prioritizing accuracy and speed.

3.6 Discussion

Due to its practical importance, method-level maintenance research has gained significant traction in recent years, making accurate method-level change history generation increasingly critical. The goal was to assess whether existing tools for generating method-level change histories produce accurate and complete results when evaluated against rigorously constructed oracles, and to identify which tool offers the best balance between accuracy and runtime efficiency. The results show that earlier evaluations based on the original *CodeShovel* and *CodeTracker* oracles were inaccurate, undermining the validity of prior assessments of tool performance (**RQ1**). To address this, two new oracles—the corrected *CodeShovel* oracle and the new *HistoryFinder* oracle—were systematically constructed, offering a more reliable benchmark that combines automated detection with expert validation.

To further address the limitations observed in both *CodeShovel* and *CodeTracker* tools, a new algorithm and tool, *HistoryFinder*, was also developed to improve both the accuracy and completeness of method change history tracking while maintaining competitive runtime performance. When evaluated against the newly constructed oracles, *HistoryFinder* consistently achieved the highest precision, recall, and F1 scores (**RQ2**) and demonstrated the most robust performance across all oracles, including those involving challenging repositories with extensive changes to the file containing the target method. Although the `Git`-based tools delivered faster execution times, this efficiency came at the cost of substantially lower

accuracy, making *HistoryFinder* the most reliable option for users prioritizing both accuracy and reasonable runtime (**RQ3**). The runtime analysis also revealed that repositories with frequent file modifications result in longer processing times for tools that rely on `JavaParser`.

Both the research community and practitioners can benefit from these contributions. Researchers can leverage these oracles to train and evaluate new algorithms or improve existing method history generation tools, especially in handling complex scenarios—such as frequent file changes, large-scale refactorings that involve method or file moves, and tracking changes in annotations or JavaDocs associated with methods. A graphical user interface (GUI) is provided that allows both researchers and practitioners to interactively run multiple tools—including *CodeShovel*, *CodeTracker*, *HistoryFinder*, *GitLineRange*, and *GitFuncName*—on their own selected Java methods. Users can independently compare outputs and choose the one best suited to their needs, whether they prioritize accuracy or runtime performance. This tool can also be integrated into a developer’s daily workflow.

In addition to the GUI, this chapter provides support for a command-line interface (CLI) that can be used to track change histories of millions of methods for conducting research on mining software repositories and understanding software evolution at the method level. For example, researchers can explore why some source methods change frequently while others remain stable, and why, when, and how bugs are introduced in source code. Additionally, *HistoryFinder* can potentially be used to track test methods, given the crucial role of test code in software quality and maintainability. The tool is also shared as a Java library for seamless integration into larger software engineering pipelines. The complete setup and usage instructions are provided with the shared repository.¹¹

3.6.1 Threats to Validity

Internal Validity. Although multiple state-of-the-art history generation tools were leveraged to construct the two oracles, the validation of change commits still involved human

¹¹<https://github.com/SQMLab/HistoryFinder>

evaluators. To mitigate this threat, two of the authors—each with over eight years of industry experience—independently conducted the validations. Nonetheless, this threat could be further reduced by involving additional evaluators. Similar to the *CodeShovel* and *CodeTracker* tools, a key limitation of the *HistoryFinder* tool is its reliance on `JavaParser`, which may fail when processing syntactically incorrect files—often introduced by inadvertent changes during development.

External Validity. Both the *CodeShovel* and *HistoryFinder* oracles were constructed using only open-source repositories, which may limit the generalizability of the findings. It remains uncertain whether the tools’ performance and accuracy would translate to closed-source or industrial settings, where code characteristics and commit practices may differ. While the *HistoryFinder* tool can be extended to support other programming languages by incorporating language-specific parsers, it is unclear whether the similarity thresholds trained for identifying method matches would remain effective outside of Java.

3.7 Conclusion and Future Work

In this chapter, the evaluation of *method-level* change history generation tools was revisited by introducing two rigorously constructed oracles. These benchmarks address key limitations in prior oracles and enable more reliable assessments of tools’ performance. The empirical analysis shows that the newly developed *HistoryFinder* tool consistently outperforms existing research-based tools such as *CodeShovel* and *CodeTracker* in terms of precision, recall, F1 score and runtime performance. Moreover, a publicly accessible, user-friendly web interface and a command-line interface are provided so that researchers and practitioners can evaluate tools on their own method samples, making the findings immediately actionable in both academic and industrial settings.

Future work will extend *HistoryFinder* to support programming languages beyond Java. It will also explore alternative parsing strategies—potentially language-independent solu-

tions—as the current language-specific parser, **JavaParser**, has proven to be a performance bottleneck. Additionally, future work will investigate alternative algorithms for detecting method similarity, including those based on machine learning techniques.

Chapter 4

Test Method Evolution: Histories, Production-to-Test Mapping, and Test Smells

The third contribution of this thesis builds on the method-history support developed in Chapter 3 and applies it to test-code evolution. Rather than examining test code only at the file or class level, this chapter analyzes how test methods evolve over time. It combines method-history generation with method-level production-to-test mapping to examine whether test methods mainly change in response to the production methods they exercise or whether they also exhibit independent maintenance patterns.

This chapter is based on the following manuscript:

- S. Islam, A. Waseem, M. N. I. Opu, and S. Chowdhury, “Understanding Method-Level Test Code Evolution,” manuscript under review for the proceedings of the 49th IEEE/ACM International Conference on Software Engineering (ICSE), Dublin, Ireland, 2027.

Abstract

Although recent advances in method-level history tracking enabled fine-grained studies of production code evolution, research on test code evolution remains constrained by two challenges: the lack of validated approaches for reconstructing test method histories and limited knowledge of the most effective techniques for method-level production-to-test mapping. This contribution addresses both challenges to enable accurate studies of test code evolution. It first evaluates state-of-the-art method history tracking tools using a manually constructed oracle of 120 test methods from 40 open-source Java projects. It then assesses static analysis-based production-to-test mapping techniques using existing benchmarks and a newly created oracle. Leveraging the most suitable approaches, it conducts a large-scale empirical study of test method evolution by comparing the revision frequencies of test methods and the production methods they exercise.

The results show that nearly 55% of test methods undergo as many or more revisions than their corresponding production methods, challenging the common assumption that test code primarily evolves in response to production code changes. Further analysis reveals that highly change-prone yet small test methods exhibit substantially higher concentrations of test smells than other test methods. In general, the results demonstrate a more accurate approach to capturing test code change history that can help future studies on test suite evolution, test deletion and obsolescence, and automated test repairs.

4.1 Introduction

Software evolution histories provide a rich source of evidence for understanding how software systems change over time. Researchers use historical data to investigate refactoring practices [149], developer collaboration patterns [150], technical debt accumulation and repayment [20, 151, 152], and the co-evolution of related software artifacts [50, 51]. Insights from these studies inform the development of maintenance models and tools, including automated refactoring [153], bug prediction [154–157], change prediction [158–161], and automated technical debt repayment [162]. Beyond research, practitioners rely on evolution histories for provenance analysis, traceability, developer onboarding, code comprehension, and expert identification [163].

Because evolution histories are typically mined from version control systems, most prior work has examined change at coarse levels of granularity, including commits, files, and classes [164]. However, researchers and practitioners increasingly require more fine-grained histories [163, 165, 166]. Class-level maintenance models, for example, are often less actionable because locating specific issues within a large class or file is difficult [156, 163, 167]. While line-level history reconstruction has been explored [168, 169], its accuracy is limited by spurious matches between syntactically similar lines [170, 171]. To address these limitations, recent research has focused on method-level history reconstruction. Tools such as *FinerGit* (JSS 2020) [172], *CodeShovel* (ICSE 2021) [163], *CodeTracker* (FSE 2022) [60], and more recently *HistoryFinder* (ICSE 2026) [173] can trace methods across renames, moves, splits, and other transformations, enabling more precise studies of change patterns, defect proneness, and technical debt survival at the method level [20, 174, 175].

These history tracking tools, however, were developed and evaluated exclusively on production methods. Consequently, test evolution research remains constrained by the lack of reliable test method histories. Pinto *et al.* [53], for example, reported challenges recovering test method histories because test methods are frequently renamed or moved. Bhatta *et al.* [56] combined custom heuristics with *RefactoringMiner* [176] and extensive manual in-

spection to identify deleted test methods, limiting scalability. Similarly, Spadini *et al.* [54] relied on an unevaluated heuristic-based history construction approach when studying the relationship between test evolution and smells. Together, these studies suggest that the absence of a validated test method history tracking remains a key obstacle for understanding test code evolution.

Reliable test method histories, however, represent only part of the solution. Understanding *why* test methods evolve also requires identifying the production methods they exercise. Without such mappings, it is difficult to determine whether a test changed in response to production code evolution or due to factors internal to the test itself, such as poor design or test smells. Method-level production-to-test mappings would also enable more precise studies of production–test co-evolution, which have largely been conducted at file or class granularity [50–52]. Furthermore, they could support finer-grained detection of obsolete tests by identifying the specific test methods affected by changes to particular production methods [53, 55–57].

To address these methodological challenges, the following two research questions are first answered.

RQ1: Can existing history tracking tools effectively track test method revision histories?

An oracle of 120 test-method revision histories from 40 popular open-source Java projects was manually constructed. State-of-the-art history tracking tools originally developed for production code are then examined to determine whether they generalize to test methods. The results show that these tools perform well on test methods without recalibration.

RQ2: What is the most accurate approach for method-level production-to-test mapping?

In addition to existing oracles, a new oracle from 20 projects was manually constructed and used with the combined benchmark to evaluate existing mapping approaches. The results show that no single approach dominates: some achieve high precision but low recall, while others favor high recall at the expense of precision.

Building on the findings from **RQ1** and **RQ2**, test code evolution and its relationship

with test smells are then revisited through the following two research questions.

RQ3: Are test methods revised less frequently? Using the most effective history tracking and mapping approaches, the revision frequencies of test methods and the production methods they exercise are compared. Contrary to the common assumption, the results show that many test methods undergo more revisions than their corresponding production methods.

RQ4: Are test smells associated with high revisions? The association between test smells and test methods that accumulate more revisions than the production methods they exercise is investigated, helping explain maintenance effort beyond what can be attributed to production-test co-evolution. Contrary to earlier findings [54], the results show that smells in small methods are more harmful than smells in large methods.

In summary, three primary contributions are made. First, the first systematic evaluation of state-of-the-art method history tracking tools on test methods is provided. Second, the existing method-level production-to-test mapping techniques are evaluated using the existing and a new manually constructed oracle. Third, the first large-scale method-level study enabled by these validated techniques is conducted, providing new empirical insights into how test methods evolve relative to production methods and the role of test smells in test evolution.

Replication Package. All data, oracles, and source code are shared to enable replication and future research.¹ This package also includes the manually annotated labels, which required approximately 180 person-hours to construct.

4.2 Related Work and Motivation

This section reviews prior research and motivates the study presented in this chapter.

¹<https://github.com/SQMLab/method-co-evolution/blob/main/replication-package.md>

4.2.1 Software Evolution

The one constant in software is change [3]. Software evolution is commonly categorized as adaptive, corrective, perfective, and preventive [4]. A substantial body of research has examined software maintenance and refactoring activities [149, 151, 153, 177]. For example, Bavota *et al.* [177] found that 42% of refactoring activities are driven by code smells. Another major research direction focuses on understanding change- and bug-prone software components [156, 158–161, 167, 174]. Such studies aim to identify defects early and reduce maintenance cost, effort, and reputational risk.

In contrast to production code, relatively few studies focused exclusively on test code evolution. A common assumption is that test code evolves primarily in response to changes in production code [52, 178], which may render tests obsolete or redundant. Consequently, much of the research has centered on production–test co-evolution [50–52], test obsolescence [55], test deletion [56], and automated test repair [57].

Zaidman *et al.* [50] investigated whether production and test code evolve synchronously or in separate phases, observing evidence of both patterns. Wang *et al.* [51] identified common factors (e.g., modified language constructs) that indicate whether a test should change following a production code modification. Similarly, Hu *et al.* [178] proposed a transformer-based approach for repairing test cases affected by production code changes. However, the authors relied on name matching to map production and test methods, an approach that suffers from low recall (as shown in **RQ2**), causing many test cases requiring repair to be missed. Another limitation in production–test co-evolution research concerns how co-evolution is identified. Most studies assume that when a test artifact and its associated production artifact are modified together, or within a short temporal window, the pair represents an instance of co-evolution. However, this assumption was found to produce a large number of false positives [52].

Pinto *et al.* [53] identified test repair as a primary driver of test evolution and showed that assertion-focused repair techniques are often insufficient for handling complex real-world

repairs. However, without any specialized tool, tracking test cases across commits proved challenging because many test methods in their dataset were renamed or moved over time. Bhatta *et al.* [56] examined why and how test methods are deleted in real-world Java applications. They found that most deleted tests are obsolete, while some others are redundant. Interestingly, deleting approximately 20% of redundant tests reduced test suite effectiveness by lowering coverage or mutation scores. Again, due to the lack of a specialized tool, the authors relied on heuristics, *RefactoringMiner* [176], and extensive manual verification for building their test deletion dataset. This multi-dimensional process not only limited scalability but may also have affected accuracy, since *RefactoringMiner* exhibits relatively low recall (54.2%) for detecting extract & move refactorings.

In a different study, Spadini *et al.* [54] found that test methods containing test smells are more change- and bug-prone than those without smells. To reconstruct test method histories, the authors assumed that two methods were identical if they shared more than 95% cosine similarity when an exact signature match could not be found. However, the accuracy of this assumption was not validated against ground truth, unlike more recent method-level history tracking techniques [60, 163, 173]. Furthermore, the study did not investigate what proportion of test method changes were induced by changes in their corresponding production methods, a question that requires accurate method-level production-to-test mapping.

Taken together, these studies suggest that accurately tracking test method histories and establishing production-to-test mappings at the method level are crucial for an accurate understanding of test code evolution.

4.2.2 Method-level history generation

Early tools such as H storage [179] and FinerGit [172] relied on Git-based mechanisms and required expensive offline preprocessing. CodeShovel [163] addressed this limitation by generating method histories on demand using a string-similarity-based tracking algorithm. However, a follow-up study by Jodavi *et al.* [60] identified limitations in both CodeShovel’s algo-

rithm and the oracle used for its evaluation. They subsequently proposed CodeTracker [60], which leverages *RefactoringMiner* [176] to track method histories and demonstrated improved accuracy on a corrected benchmark. More recently, Islam *et al.* [173] revisited both method-history generation and its evaluation. They showed that existing tools were assessed using inaccurate ground-truth oracles and constructed a new, systematically developed benchmark for evaluation. Using this benchmark, they reevaluated prior approaches and introduced *HistoryFinder*, which achieved the best overall trade-off between tracking accuracy and execution time.

Unfortunately, these high-performing tools—CodeShovel, CodeTracker, and HistoryFinder—were developed and evaluated exclusively on production methods. As test code differs substantially from production code in many aspects [50, 180, 181], it remains unclear whether the string-based matching heuristics and thresholds [163, 173] that perform well for production methods generalize to test methods without recalibration.

4.2.3 Method-level production-to-test mapping

Linking production and test code is essential for regression test selection, traceability, co-evolution analysis, and test maintenance [62–64, 182]. However, establishing these links is challenging, particularly at the method level [62]. Test methods often invoke helper, setup, assertion, and utility methods in addition to the production methods they validate, causing static and dynamic analyses to incorrectly identify incidental calls as tested methods [63, 64].

Van Rompaey and Demeyer [61] found that simple heuristics such as *Naming Conventions (NC)*, which matches test and production method names, and *Last Call Before Assert (LCBA)*, which identifies the last production method invoked before an assertion, achieve high accuracy. However, these approaches are less effective when naming conventions are inconsistent or when a test exercises multiple production methods [63, 182]. For example, LCBA can incorrectly identify helper methods as the tested method due to assertion proximity. White *et al.* [62] evaluated several additional linking techniques, including *Naming*

Conventions Contains (NCC), which relaxes exact name matching; *Longest Common Subsequence* using both names for normalization (LCS-B) or only the production name (LCS-U); and *Levenshtein distance (Leven)*. They also explored *Tarantula* [183], which applies fault-localization principles to identify tested methods, and *Term Frequency–Inverse Document Frequency (TFIDF)*, which treats tests as documents and production methods as terms. Using a manually constructed oracle from three open-source Java projects, they found that a combined approach incorporating all techniques achieved the highest accuracy. In a later study using an expanded benchmark, the authors showed that incorporating dynamic traces substantially improves the accuracy of naming-based approaches [63].

Using dynamic traces, however, is challenging because projects must compile and tests must be executable, limiting scalability for mining software repositories (MSR) studies that analyze large numbers of projects [64]. To address this limitation, Sun *et al.* [64] proposed *TestLinker*, a purely static approach that leverages semantic correlation learning and a pre-trained code model. *TestLinker* was evaluated on an extended benchmark that added two projects to the dataset used by White *et al.* [63].

Prior research showed that the relative effectiveness of competing techniques often depends on the benchmark used for evaluation [174, 184], and a similar phenomenon is observed here. For example, the NC heuristic achieved precisions of 100%, 100%, and 93% in [61, 63, 64], respectively, while the corresponding recalls were 100%, 11%, and 12%. Such variation makes it difficult to determine the most effective approach. Therefore, before applying production-to-test mapping to the 110 projects studied here, existing techniques are systematically reevaluated using a new benchmark of 20 open-source projects, substantially larger than the benchmarks of 3 to 6 projects used in prior studies [62–64].

4.3 Experimental Setup

4.3.1 Project selection

For RQ1, where tools for constructing test method histories are compared, all 40 projects used in the most recent study, *HistoryFinder* [173], were selected. These 40 projects subsume the 20 projects previously used to evaluate both *CodeShovel* [163] and *CodeTracker* [60]. For RQ2, where the ground truth for production-to-test mappings is constructed, the annotation effort was substantially greater than for RQ1. To keep the manual effort manageable, the 20 projects used in *CodeShovel* and *CodeTracker* studies were used, rather than all 40 projects from *HistoryFinder*. Despite this reduction, the benchmark is considerably larger than those used in prior work. For example, the state-of-the-art *TestLinker* [64] was evaluated on only six projects. For RQ3 and RQ4, which rely exclusively on quantitative analysis, all projects used in recent method-level evolution studies [163, 173, 174, 185] were combined, resulting in a dataset of 110 projects.

4.3.2 Test and production method detection

To identify all the production and test files, the standard test and production source-root conventions were initially used. For example, in both Maven and Gradle Java projects, the default production and test source roots are `src/main/java` and `src/test/java`, respectively. Since projects can override or extend these defaults, build configuration files were also analyzed to identify project-specific source-root declarations, such as the `sourceDirectory` and `testSourceDirectory` elements in Maven `pom.xml` files and the `sourceSets` properties in Gradle build files. Finally, each project’s layout was manually inspected, and some projects were found to place test files under the default production source root, especially when an entire module is intended for testing. In these cases, the corresponding files were considered test files.

After identifying the test files, methods within the file were classified as either test meth-

ods or test-helper methods. Following prior work that detects JUnit tests using framework-specific annotations and naming conventions [50, 186], methods annotated with test annotations were identified first. These include, for example, `org.junit.jupiter.api.Test` for JUnit, `org.testng.annotations.Test` for TestNG, and `net.jqwik.api.Property` for jqwik. To detect pre-JUnit 4.x tests, public, parameterless `void` methods whose names begin with `test` were identified as test methods when their declaring class extends `junit.framework.TestCase` or one of its descendants. All other methods declared in test files are classified as test-helper methods—methods that are not directly testing any production methods. This identification was important as these methods were filtered out during production-to-test mapping.

Files not identified as test files were labelled as production files, and all methods in those files were extracted as production methods. Similar to prior works [159, 187, 188], `getters` and `setters` were filtered out; these methods are often auto-generated and create noise in software maintenance research.

4.4 Approach and Results

The approach and results of each RQ are discussed below.

4.4.1 RQ1: Evaluation of existing history generation tools

To advance research on test code evolution, it is necessary to determine whether tools developed for production code generalize to test code or require retraining and threshold recalibration.

Approach

The *CodeShovel* [163] and *CodeTracker* [60] studies used the same 20 popular open-source Java projects to construct their evaluation oracles. *HistoryFinder* [173] later expanded this

benchmark by adding 20 additional projects. All 40 projects were used to construct a new oracle for evaluating test-method history generation. From each project, three test methods were randomly selected, resulting in 120 test-method histories, slightly more than the original *CodeShovel* [163] study, which used 100 methods as test data.

To construct the ground truth, the methodology of *HistoryFinder* [173] was followed. Specifically, candidate change commits were collected from six history-generation tools: *HistoryFinder*, *CodeShovel*, *CodeTracker*, *IntelliJ*, *GitLineRange*, and *GitFuncName*. The union of all commits reported by these tools was then formed, which offers almost perfect recall [173], but contains many false positives. This is why, similar to [173], the first and second authors, graduate students in software engineering with over 10 years and 1 year of industry experience, respectively, independently reviewed each of the generated commits to filter out the false positives, requiring ≈ 80 person-hours. The agreement score was near perfect (Cohen’s Kappa of 0.98). The disagreements in 10 commits were subsequently resolved through in-person discussion. Following prior studies [163, 173], any selected test method with fewer than three revisions was replaced by another randomly selected test method from the same project with at least three revisions.

Similar to *HistoryFinder* [173], the tools’ performance was assessed at both the commit and method levels. At the commit level, true positives, false positives, and false negatives were aggregated across all test methods before computing precision, recall, and F1 score. At the method level, these metrics were computed separately for each test method and their averages were reported. The execution time (the average of five runs) required to generate each method history was also measured. Experiments were conducted on a machine running Ubuntu 22.04 LTS with 32 GB of RAM, an Intel(R) Core(TM) i7-8700 CPU @ 3.20 GHz (12 cores), and Java 21 (x86). To avoid cross-run caching effects, each tool was executed independently for every method. Similar to [173], *IntelliJ* was excluded from the runtime comparison because its histories must be obtained manually through the IDE’s built-in history-tracking feature.

Result

Table 4.1: Accuracy of the method-history tracking tools. TP denotes correctly identified change commits, FP denotes incorrectly included commits, and FN denotes missed change commits.

Tool	<i>Commit-Level</i>						<i>Method-Level</i>		
	TP	FP	FN	Precision	Recall	F1	Precision	Recall	F1
CodeShovel	644	3	16	99.54	97.58	98.55	99.35	97.82	98.27
CodeTracker	667	11	18	98.38	97.37	97.87	97.56	96.57	97.72
HistoryFinder	721	5	1	99.31	99.86	99.59	99.63	99.65	99.60
IntelliJ	549	180	171	75.31	76.25	75.78	87.90	76.96	79.72
GitLineRange	343	58	350	85.54	49.49	62.71	92.56	51.40	61.80
GitFuncName	86	8	611	91.49	12.34	21.74	98.90	10.69	11.35

Table 4.1 shows that all three research-based tools substantially outperform the *IntelliJ* and *Git-based* baselines. At the commit level, *HistoryFinder* achieves the best overall accuracy, attaining the highest recall (99.86%) and F1 score (99.59%). *CodeShovel* achieves the highest precision (99.54%) and also performs strongly overall, with an F1 score of 98.55%. *CodeTracker* follows closely with an F1 score of 97.87%. The method-level results exhibit a similar trend.

Table 4.2 and Figure 4.1 compare the runtime performance of these tools. The *Git-based* approaches achieve the lowest runtimes; however, this advantage largely stems from terminating history tracing much earlier than the other tools, resulting in substantially lower recall. Among the research-based approaches, *HistoryFinder* is the most efficient, with the lowest mean (1.47 seconds) and median (0.63 seconds) runtimes. *CodeTracker* exhibits the greatest runtime variability and the highest maximum runtime (943.93 seconds), primarily due to its reliance on *RefactoringMiner* [176] for tracking refactorings. In contrast, both *HistoryFinder* and *CodeShovel* rely on lightweight string-based similarity techniques, resulting in lower and more stable runtimes.

Table 4.2: Runtime of the history generation tools in seconds.

Tool	Mean	Median	Max
HistoryFinder	1.47	0.63	37.87
CodeShovel	2.52	1.22	42.02
CodeTracker	26.53	1.14	943.93
GitLineRange	1.18	0.28	20.81
GitFuncName	0.32	0.18	3.95

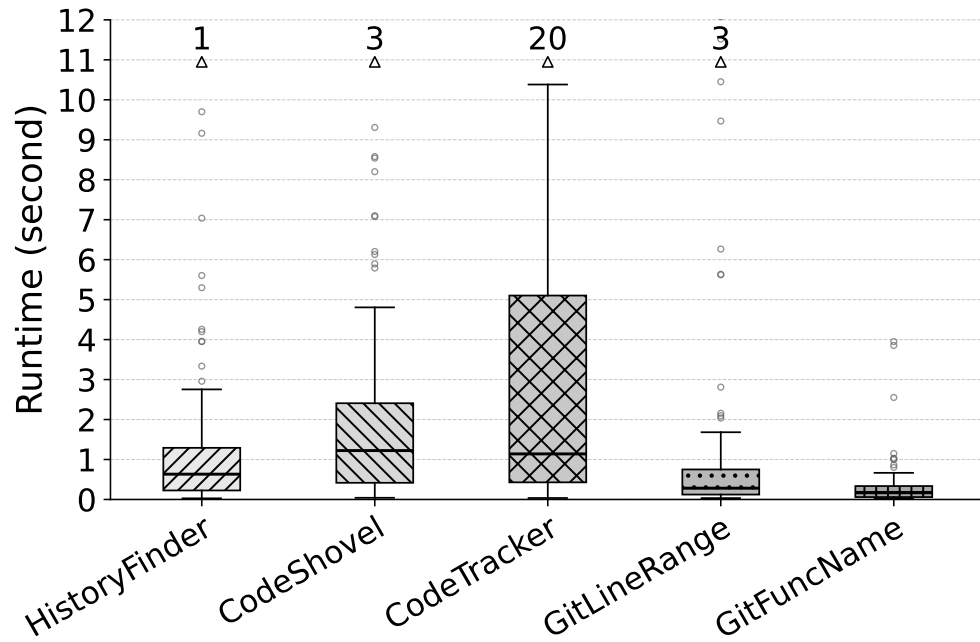


Figure 4.1: Runtime of the studied tools. The y-axis is capped at 12s for readability; triangles indicate values above the limit, with numbers showing their counts.

RQ1 summary: Considering both accuracy and runtime, *HistoryFinder* is the best tool for tracking test method change history. Overall, the results are encouraging, as they suggest researchers can use existing tools directly without the cost of retraining them for test methods.

4.4.2 RQ2: Production-to-test linking

To support large-scale MSR studies on test maintenance [186], this section investigates which static production-to-test mapping approach provides the highest accuracy. This is done by reevaluating existing approaches on both prior and newly constructed benchmarks and by introducing two additional heuristic-based techniques.

Approach

To improve evaluation reliability, a new benchmark was constructed using the 20 projects employed in previous method-history tracking studies [60, 163, 173]. More projects were not included to keep the manual annotation manageable. For each project, the Java source code was parsed using the JavaParser Symbol Solver² to construct a static call graph. For every test method, all unique production methods reachable through direct calls formed the candidate set. Then, 20 test methods per project were randomly selected, yielding 400 test methods for manual annotation.

The first and second authors independently mapped production methods to their corresponding test methods. This process required ≈ 100 person-hours in total. Disagreements on 46 test methods were resolved through discussion. During annotation, a production method was considered a positive link only if it represented the behavior being tested. Setup calls preceding the tested behavior and verification calls used solely to inspect results were not labeled as positive links. This distinction is not always reflected in existing

²<https://github.com/javaparser/javaparser/tree/master/javaparser-symbol-solver-core> last accessed: June 28, 2026

benchmarks. For example, in the *Commons Lang* project, `testSetSizeStartText` exercises `setSizeStartText(null)` and verifies the outcome using `getSizeStartText()`.³ Therefore, the getter should not be considered the production method under test, although it is labeled as such in early datasets [63]. However, some test methods were indeed observed to intentionally test multiple production methods. In such cases, all relevant production methods were labeled as positive links. Consequently, the benchmark contains 451 confirmed production–test links rather than exactly 400, in addition to 3,816 non-links. The non-links refer to methods that were invoked by the 400 test methods but were not actually tested.

At the tool level, previous approaches were found to rely on different call-graph construction techniques and generally did not leverage the recent JavaParser Symbol Solver. Therefore, using the original replication packages and author support (e.g., via email communication for *TestLinker* [64]), existing approaches [62, 64] were modified to use JavaParser Symbol Solver. As shown later, this modification substantially improves the performance of all existing techniques.

Besides reevaluating prior approaches, two simple techniques are also introduced: *One Method Call (OMC)* and *OMC+NC*. *OMC* removes helper methods and irrelevant external calls (e.g., Java standard-library calls) from the candidate set and links a test method to a production method only when a single candidate remains. The intuition is that if only one production method is exercised, it is likely the method under test. While this strategy favors high precision, its recall may be limited. To improve coverage, *OMC+NC* takes the union of links identified by *OMC* and the Naming Conventions (*NC*) heuristic, combining structural and naming evidence.

Following prior production-to-test mapping studies [62–64], approaches are evaluated at the level of unique production-test method pairs. A correctly predicted link is counted as a true positive (*TP*), an incorrect prediction as a false positive (*FP*), and a missed confirmed link as a false negative (*FN*). Precision, recall, and F1 score are reported, and

³<https://github.com/apache/commons-lang/blob/425d8085cfcaab5a78bf0632f9ae77b7e9127cf8/src/test/java/org/apache/commons/lang3/builder/ToStringStyleTest.java#L91> last accessed: June 28, 2026

Table 4.3: Production-to-test link prediction performance by dataset for all approaches.

Dataset	Strategy	TP	FP	FN	Precision	Recall	F1	MAP	AUC
TestLinker [64]	LCBA	244	162	90	0.60	0.73	0.66	0.64	–
	LCS-B	250	160	84	0.61	0.75	0.67	0.70	0.76
	LCS-U	227	83	107	0.73	0.68	0.70	0.70	0.81
	Leven	181	59	153	0.75	0.54	0.63	0.62	0.77
	NC	48	3	286	0.94	0.14	0.25	0.20	–
	NCC	122	20	212	0.86	0.37	0.51	0.41	–
	Tarantula	284	322	50	0.47	0.85	0.60	0.65	0.73
	TFIDF	272	235	62	0.54	0.81	0.65	0.63	0.73
	Combined	233	58	101	0.80	0.70	0.75	0.74	0.86
	TestLinker	192	47	142	0.80	0.57	0.67	0.66	–
	OMC	69	4	265	0.95	0.21	0.34	0.30	–
	OMC+NC	99	7	235	0.93	0.30	0.45	0.41	–
New	LCBA	210	472	241	0.31	0.47	0.37	0.39	–
	LCS-B	241	592	210	0.29	0.53	0.38	0.44	0.45
	LCS-U	234	410	217	0.36	0.52	0.43	0.48	0.51
	Leven	168	174	283	0.49	0.37	0.42	0.37	0.47
	NC	25	1	426	0.96	0.06	0.10	0.06	–
	NCC	132	46	319	0.74	0.29	0.42	0.30	–
	Tarantula	302	998	149	0.23	0.67	0.34	0.40	0.51
	TFIDF	291	870	160	0.25	0.65	0.36	0.40	0.51
	Combined	235	301	216	0.44	0.52	0.48	0.48	0.53
	TestLinker	221	126	230	0.64	0.49	0.55	0.53	–
	OMC	57	14	394	0.80	0.13	0.22	0.14	–
	OMC+NC	76	15	375	0.84	0.17	0.28	0.19	–
All	LCBA	454	634	331	0.42	0.58	0.48	0.48	–
	LCS-B	491	752	294	0.40	0.63	0.48	0.54	0.57
	LCS-U	461	493	324	0.48	0.59	0.53	0.56	0.62
	Leven	349	233	436	0.60	0.44	0.51	0.46	0.59
	NC	73	4	712	0.95	0.09	0.17	0.11	–
	NCC	254	66	531	0.79	0.32	0.46	0.34	–
	Tarantula	586	1320	199	0.31	0.75	0.44	0.50	0.59
	TFIDF	563	1105	222	0.34	0.72	0.46	0.48	0.59
	Combined	468	359	317	0.57	0.60	0.58	0.57	0.66
	TestLinker	413	173	372	0.70	0.53	0.60	0.57	–
	OMC	126	18	659	0.88	0.16	0.27	0.20	–
	OMC+NC	175	22	610	0.89	0.22	0.36	0.27	–

ranking performance is additionally evaluated using MAP and AUC. MAP computes the average precision of ranked candidate production methods for each test and then macro-averages across tests. AUC summarizes the overall precision-recall tradeoff of scoring-based approaches and is not applicable to binary-only techniques such as *NC*.

Result

Table 4.3 summarizes the results for all approaches across all benchmarks. The TestLinker dataset [64] extends the earlier datasets [62, 63] by adding two projects, yielding a total of six projects. The new benchmark contains 20 projects, while the *All* benchmark combines both datasets.

Comparing the results on the *TestLinker* benchmark with those originally reported by Sun *et al.* (Table III, static-analysis approaches [64]) shows substantial improvements across all baselines (e.g., *NC*, *NCC*, *LCBA*, and *LCS-B*). For example, the F1 score of *LCS-U* increased from 0.37 to 0.70, while the F1 score of *TFIDF* increased from 0.33 to 0.65. These gains are attributable to the use of JavaParser Symbol Solver for accurate call graph construction. We also observe that most approaches perform worse on the new benchmark than on the *TestLinker* benchmark. This decline is likely due to the larger and more diverse set of projects, which increases the probability of having more cases where developers do not consistently follow naming conventions—a key signal leveraged by most static-analysis-based mapping techniques.

Overall, no single approach consistently performs best across all benchmarks and evaluation metrics. Instead, the approaches fall into three broad categories: high-precision approaches, recall-oriented approaches, and approaches that maintain balance between precision and recall. For example, the *NC* and *OMC* achieve very high precision but suffer from low recall. In contrast, *Tarantula* consistently achieves high recall at the cost of substantially lower precision. Approaches such as *Combined* and *TestLinker* provide the best balance between precision and recall and consequently achieve the highest F1 scores across

benchmarks.

These findings indicate that the choice of linking approach should depend on the downstream analysis. Conservative approaches such as *NC* are preferable when false positives are costly. *OMC* and *OMC+NC* provide a middle option for increasing coverage while remaining precision-oriented. *TestLinker* and *Combined* are preferable when a stronger balance between precision and recall is needed. *Tarantula* and *TFIDF* are appropriate only when maximizing link recovery is more important than avoiding incorrect links.

RQ2 summary: There is no single best approach for linking production and test methods. The choice of linking technique should be guided by the requirements of the downstream analysis. Approaches such as *NC* and *OMC+NC* are preferable when high precision is critical, whereas *Tarantula* is better suited for scenarios that prioritize high recall and can tolerate lower precision.

4.4.3 RQ3: Revision in test and production methods

Prior studies have shown that test code does not evolve as consistently as production code [50, 189]. Test maintenance often lags behind production changes, and many production-code modifications are performed without corresponding updates to the test suite. For many projects, fewer than 15% of commits involve test maintenance [189], and even popular open-source projects may experience limited testing activity [190]. Furthermore, when test methods are modified, such changes are often assumed to be triggered by changes in the corresponding production methods [52, 178]. While test code can evolve alone due to its own internal problems, such as smells, that is also true for production code. Collectively, these observations suggest that test code should evolve less frequently than production code. In this section, accurate method-level history tracking (RQ1) and mapping (RQ2) are leveraged to empirically evaluate this assumption by comparing the revision characteristics of test and production methods.

Approach

As *HistoryFinder* achieved the best performance for tracking both production code [173] and test code (RQ1), it was used to collect revision histories for all methods in the 110 projects described in Section 4.3. After filtering helper test methods as well as getter and setter methods from the production code (Section 4.3), the revision histories of 1,760,423 production methods and 567,904 test methods were collected. Although *HistoryFinder* reports a revision when the containing file of a method is renamed, even if the method itself remains unchanged, such revisions were excluded from the dataset. The complete project-level statistics are available in the replication repository (`data/rqs/rq3/method-count.csv`).

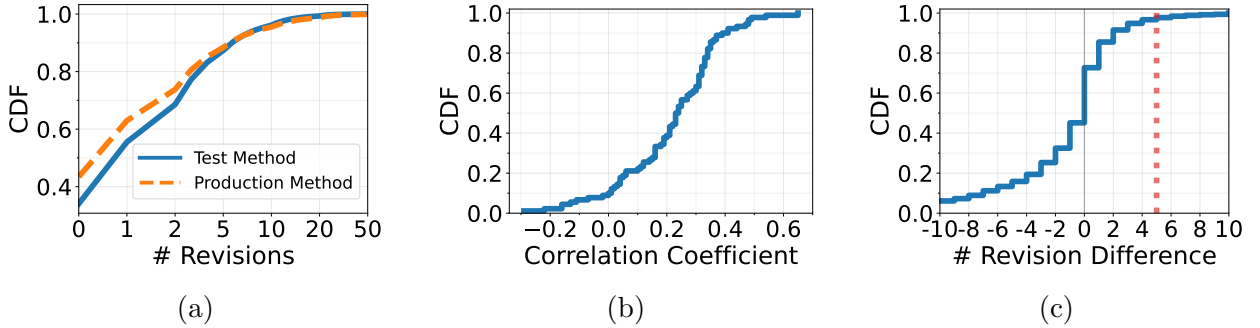


Figure 4.2: Figure (a) shows the CDF of the number of revisions for both production and test methods. To enhance readability, the x-axis is restricted to 50. Figure (b) shows the CDF of correlation coefficients between the number of revisions in test methods and their corresponding production methods. Figure (c) shows the CDF of differences in revisions between test methods and their corresponding production methods. It is important to note that all test and production methods were used for Figure (a), while only mapped examples were used for (b) and (c).

When production-to-test mappings were required, precision was prioritized over recall because reliable links are needed for reliable analysis. Also, given the large number of projects and methods, even a lower-recall approach can produce a sufficiently large sample for reliable statistical analysis, provided that the mappings are accurate. Therefore, based on the findings of RQ2, the *OMC+NC* approach was adopted, which provides substantially higher recall than *NC* while incurring a small reduction in precision. For example, for the *all* dataset, the precisions of *NC* and *OMC+NC* are 0.95 and 0.89, respectively. However, the recall of *NC* is only 0.09 in contrast to 0.22 of *OMC+NC*.

Result

Figure 4.2 (a) shows the cumulative distribution function (CDF) of revisions for production and test methods after aggregating data across all projects. Surprisingly, test methods underwent more revisions than production methods when comparing low change-prone methods (0 to 2 revisions). For example, more than 62% of test methods were revised at least once, compared with approximately 56% of production methods. The non-parametric Wilcoxon rank-sum test [20, 191] shows this difference to be significant ($p\text{-value} \leq 0.05$). The same comparison was also performed at the project level, showing that test methods were revised more frequently than production methods in 65 projects, with statistically significant differences in 55 projects. The corresponding Cliff’s delta effect sizes [192] were negligible, small, medium, and large in 41.8%, 40%, 12.7%, and 5.5% of the projects, respectively. In contrast, production methods were revised more frequently in 43 projects, with statistically significant differences in 36 projects. The full table is available with the replication package (</data/rqs/rq3/revision-statistics.pdf>).

For each project separately, using the mapped production-to-test examples, Kendall’s τ correlation coefficient [193] was calculated between the number of revisions of test methods and the number of revisions of their corresponding production methods. Figure 4.2 (b) shows that, for 60% of the projects, the coefficients are below 0.3. This finding contrasts with the common assumption that test methods are revised primarily in response to changes in their corresponding production methods. Under this assumption, one would expect co-evolution between production and test methods and, consequently, a strong correlation in their revision frequencies.

To investigate more, the mapped test and production samples were used again to calculate the revision difference between test methods and their corresponding production methods. Figure 4.2 (c) shows that, in only 45% of the cases, production methods were revised more frequently than their test methods. In 26% of the cases, the number of revisions was identical. For the remaining 29% cases, test methods were revised more than their corresponding

production methods. Notably, more than 3% of test methods were revised at least five more times than their corresponding production methods.

RQ3 summary: Leveraging accurate test method history tracking and production-to-test mappings, the results show that in many projects, test methods undergo more revisions than their corresponding production methods.

4.4.4 RQ4: Test smells and revisions

The previous analysis shows that revisions to test methods are not necessarily driven solely by changes in the production methods they exercise. Test methods may also evolve because of internal maintainability issues. Test smells provide a useful lens for examining this possibility, as they capture potentially problematic characteristics of test code [54, 194, 195]. Prior work has shown that test smells are most often introduced in the initial version of a test method [196]. Therefore, this section investigates whether the presence of test smells in the first version of a test method is associated with a high number of future revisions.

Approach

For smell detections, *JNose* [197] was used. Using rule-based detectors derived from established test-smell definitions and detection rules from prior work [196, 198], *JNose* detects different types of test smells and provides method-level smell locations, making it appropriate for this analysis. To understand the association of test smells with method evolution, the test methods are divided into three groups and labeled for easy reference for subsequent analysis.

- **NTR:** These are methods that were not revised more than their corresponding production methods (normal test revision).
- **MTR:** These methods were revised more than their production methods (more test revisions).

- **HTR:** A subset of *MTR*, these methods were revised at least five more times than their production methods (high test revisions).

A test method may exercise multiple production methods and, as a result, may receive different group assignments depending on which linked production method is used. To reduce ambiguity, such methods were filtered out for this RQ. Also, to ensure sufficient project-level coverage, projects for which the strategy establishes at least 30 links were included. After applying this threshold, the resulting sample includes 71,231 *NTR* methods from 90 projects, 23,494 *MTR* methods from 88 projects, and 3,274 HTR methods from 74 projects.

The following metrics were used for comparing these three groups of methods.

- **Percent (%)**: For each individual smell, its prevalence was calculated in the three groups of methods. The difference in percent (Δ **pp**) was also calculated.
- **Odds ratio (OR)**: Suppose the odds ratio is calculated by using *NTR* and *HTR* groups. Let a be the number of HTR methods with a smell, b the number of HTR methods without the smell, c the number of NTR methods with the smell, and d the number of NTR methods without the smell. The odds ratio is computed as $OR = (a/b)/(c/d) = ad/bc$, so values above one indicate that the smell is more prevalent among HTR methods. The conditional maximum likelihood estimate of the odds ratio and its exact 95% confidence interval are reported.
- **Fisher’s exact tests (p_{BH}) [199]**: Two-sided Fisher’s exact tests were also used because the group sizes are unequal and several smells are rare. The Benjamini–Hochberg [200] procedure was also applied across the individual-smell comparisons to control the false discovery rate at 0.05.
- **Mantel-Haenszel (MH OR and MH p_{BH}) [201]**: Projects may differ in their testing practices and baseline smell prevalence. As a sensitivity analysis, a Mantel-Haenszel odds ratio was additionally estimated for each smell while treating projects as strata, and the same multiple-comparison correction.

Result

Table 4.4: The top part of the table shows results for comparison between the *NTR* and *MTR* methods. For example, **NTR %** reports the percent of methods in the *NTR* group with a particular smell. The bottom part shows the comparisons between the *NTR* and *HTR* methods for a few smells. The full table for the bottom part is available (</data/rqs/rq4/test-smell-difference.pdf>).

Test smell	NTR %	MTR %	Δ pp	OR [95% CI]	p_{BH}	MH OR	MH p_{BH}
Assertion Roulette	13.5	20.1	+6.6	1.61 [1.55, 1.68]	< .05	1.48	< .05
Verbose Test	2.4	6.2	+3.8	2.67 [2.49, 2.86]	< .05	2.30	< .05
Magic Number Test	7.0	10.4	+3.3	1.52 [1.45, 1.60]	< .05	1.48	< .05
Exception Handling	3.0	4.9	+1.9	1.69 [1.57, 1.82]	< .05	1.70	< .05
Conditional Test Logic	2.8	4.7	+1.9	1.72 [1.59, 1.85]	< .05	1.71	< .05
Eager Test	4.5	6.3	+1.9	1.44 [1.36, 1.54]	< .05	1.51	< .05
Duplicate Assert	2.8	4.5	+1.7	1.64 [1.52, 1.77]	< .05	1.45	< .05
Unknown Test	5.2	5.6	+0.4	1.09 [1.02, 1.16]	< .05	0.93	0.057
Mystery Guest	0.5	0.9	+0.4	1.73 [1.46, 2.06]	< .05	1.63	< .05
Sensitive Equality	1.1	1.5	+0.4	1.32 [1.17, 1.50]	< .05	1.46	< .05
Resource Optimism	0.4	0.7	+0.3	1.71 [1.40, 2.07]	< .05	1.61	< .05
Sleepy Test	0.2	0.5	+0.3	2.31 [1.78, 2.98]	< .05	2.76	< .05
Print Statement	0.3	0.4	+0.1	1.19 [0.92, 1.51]	0.173	2.65	< .05
Redundant Assertion	0.2	0.2	+0.1	1.34 [0.97, 1.83]	0.084	1.96	< .05
Empty Test	0.1	0.1	+0.0	1.31 [0.67, 2.45]	0.420	2.03	0.072
Lazy Test	9.2	8.9	-0.3	0.96 [0.91, 1.01]	0.106	0.96	0.106
Ignored Test	2.0	1.1	-0.9	0.54 [0.47, 0.61]	< .05	0.40	< .05
Test smell	NTR %	HTR %	Δ pp	OR [95% CI]	p_{BH}	MH OR	MH p_{BH}
Assertion Roulette	13.5	30.0	+16.5	2.75 [2.53, 2.98]	< .05	2.42	< .05
Verbose Test	2.5	14.3	+11.9	6.66 [5.94, 7.46]	< .05	5.47	< .05
Exception Handling	3.0	8.4	+5.5	3.01 [2.62, 3.45]	< .05	3.08	< .05
Conditional Test Logic	2.8	8.3	+5.5	3.15 [2.74, 3.61]	< .05	2.78	< .05
Duplicate Assert	2.8	7.6	+4.8	2.86 [2.47, 3.29]	< .05	2.44	< .05
Sleepy Test	0.2	1.4	+1.2	6.97 [4.78, 9.96]	< .05	9.57	< .05

The top part of Table 4.4 shows that, with the exception of a few smell types, such as *Redundant Assertion* and *Lazy Test*, *MTR* methods are more associated with smells than *NTR* methods. For example, 20.1% of *MTR* methods contain the *Assertion Roulette* smell, compared to 13.5% of *NTR* methods. The odds ratio (**OR**) indicates that *MTR* methods are 1.6 times more likely to contain this smell than *NTR* methods. These differences are also statistically significant, as both p_{BH} and **MH** p_{BH} are below 0.05.

This trend becomes even more pronounced (bottom part of the table) when comparing *HTR* methods—those revised at least five more times than their corresponding production

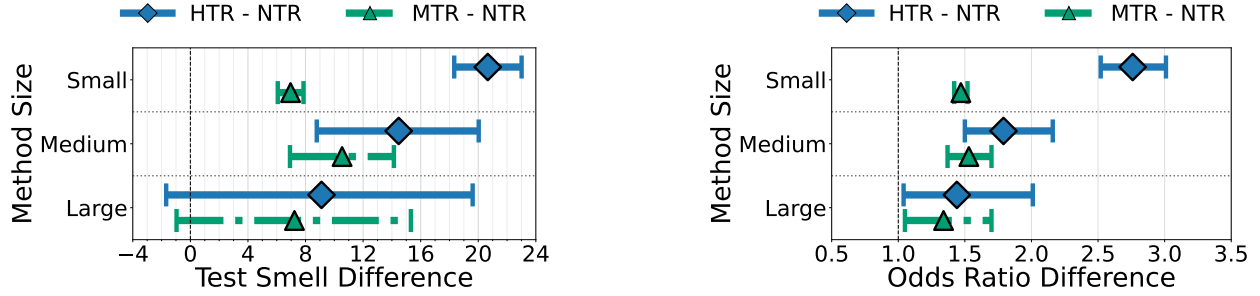


Figure 4.3: Size-based analysis. Figure (a) shows the difference in percent with 95% confidence interval, and Figure (b) shows the odds ratio (OR). The comparison is for *MTR* and *NTR* methods, as well as *HTR* and *NTR* methods. The selected 11 types of smells are joined together and considered as one smell.

methods—with *NTR* methods. For most smell types, both the odds ratio and the difference in percent (Δ **pp**) increase substantially. In particular, the odds ratios for the *Verbose Test* and *Sleepy Test* smells indicate that these smells are more than six times as likely to occur in *HTR* methods as in *NTR* methods. Overall, Table 4.4 suggests that test methods that evolve more frequently than their corresponding production methods tend to contain more test smells than less change-prone methods.

The potential confounding effect of method size was also examined when interpreting the relationship between test smells and method evolution. Prior studies have shown that many code metrics and smells lose their predictive power for software quality and maintainability once size is controlled for, as size is often strongly correlated with these metrics [202–204]. For this analysis, noise was minimized by excluding smell types that did not exhibit a consistent statistically significant difference in Table 4.4. Specifically, *Unknown Test*, *Print Statement*, *Redundant Assertion*, *Lazy Test*, *Empty Test*, and *Ignored Test* were excluded. Using the remaining 11 smell types, Kendall’s τ correlation coefficient was computed between size and the number of test smells per method. The resulting coefficient of 0.28 suggests that method size alone does not explain the observed relationship between test smells and method evolution. This apparent discrepancy with previous studies [202–204] is unsurprising, as those studies primarily reported strong correlations between size and software metrics at the class or file level. In contrast, subsequent work has shown that such correlations are

considerably weaker at the method level [160, 205], which is the granularity of this analysis.

To further investigate the impact of method size, following the methodology of Spadini *et al.* [54], test methods were categorized into three size groups: small ($size < 30$), medium ($30 \leq size \leq 60$), and large ($size > 60$). Figure 4.3 presents the results of this size-stratified analysis. Consistent with the earlier findings, both the difference in percent (Δ **pp**) and the odds ratio are larger when comparing *HTR* methods with *NTR* methods than when comparing *MTR* methods with *NTR* methods. More importantly, the association between test smells and method evolution is strongest for small methods: both the percent difference and the odds ratios are considerably larger for small methods than for medium or large methods. This observation contradicts earlier findings [54] that test smells in large methods are more harmful than smells in smaller methods.

RQ4 summary: Smells are more associated with test methods that evolve more than their corresponding production methods. This association becomes even stronger for test methods that are revised at least five more times than their production methods. Furthermore, smells in small test methods show more predictive power than smells in large methods, contradicting earlier findings.

4.5 Discussion

Two key challenges that limit test-code evolution research were addressed: reliable test-method history tracking and method-level production-to-test mapping. The first finding (**RQ1**) is encouraging: history-generation tools developed for production methods generalize well to test methods without any threshold recalibration. In particular, *HistoryFinder* [173] achieved the best overall performance in both accuracy and runtime. In contrast, the evaluation of production-to-test mapping (**RQ2**) showed that no approach consistently achieves both high precision and high recall. Thus, the choice should be guided by the downstream objective. For example, precision-oriented approaches suit studies requiring reliable links,

whereas recall-oriented approaches better support applications prioritizing broader coverage.

These validated techniques enabled a more accurate investigation of test method evolution. The results show (**RQ3**) that nearly one-third of mapped test methods are revised more often than the production methods they exercise, while the correlation between production and test revisions is generally weak. This suggests that much of test maintenance cannot be explained solely by production-test co-evolution and that test code should be treated as a first-class software artifact with its own maintenance dynamics rather than merely a passive consequence of production-code changes.

The test-smell results provide one explanation for this additional maintenance effort. Test methods that evolve more than their corresponding production methods contain more test smells (**RQ4**). Surprisingly, the association is strongest in small test methods, contradicting Spadini et al. [54]. This difference likely stems from more accurate method-history tracking, production-to-test method alignment, and a larger benchmark (110 vs. 10 projects). The stronger impact of test smells in smaller methods underscores the importance of evaluating software quality indicators at the method level rather than extrapolating conclusions from file/class-level analyses.

For researchers, the findings have several implications. The ability to track test method evolution alongside production changes may help more reliable studies of test obsolescence by precisely identifying when a test becomes outdated relative to the production change that invalidated it. It may also support the study of flaky tests caused by evolutionary mismatch. In addition, it can improve regression testing research by enabling fine-grained identification of which tests should be re-run or updated after specific production changes. For test repair, it can help pinpoint the exact divergence point and derive minimal corrective fixes. Finally, it enables more accurate long-term test maintenance modeling by supporting measurement of technical debt accumulation, smell persistence, and maintenance effort over time.

For practitioners, the results indicate that test maintenance represents a substantial portion of long-term maintenance effort and deserves no less engineering attention than

production code. Organizations should therefore monitor the evolution of test code independently instead of assuming that production-code metrics adequately reflect testing effort. The strong relationship between test smells and highly evolving tests also suggests practical opportunities for maintenance prioritization [206].

4.6 Threats to Validity

Construct Validity. The analyses relied on method-level history generation, production-to-test mapping, and test-smell detection. Some of the relevant threats were mitigated by first evaluating existing history-generation and mapping techniques on manually constructed oracles and then using the best-performing approaches in the subsequent analyses.

Internal Validity. The change-history and production-to-test mapping oracles involved human judgment. To mitigate this threat, two authors independently performed the annotations and resolved disagreements through discussion.

External Validity. The dataset includes 110 open-source Java projects. Although this is substantially larger than the datasets used in prior work, the findings may not generalize to closed-source systems and other programming languages. Also, due to focusing on a precision-oriented approach for mapping between production and test, many examples were missed, which may harm generalizability.

Conclusion Validity. Appropriate non-parametric statistical tests, effect sizes, confidence intervals, and false-discovery-rate correction were used where applicable. However, as with many observational studies, the findings confirm associations rather than guaranteed causal relationships.

4.7 Conclusion

In this chapter, a comprehensive analysis of method-level test code evolution was presented. The results show that existing history-generation tools generalize well to test methods with-

out recalibration. For production-to-test mapping, no single approach consistently achieves both high precision and high recall. Using the most suitable techniques, the results showed that test methods often evolve independently of their corresponding production methods and that highly evolving small test methods are more likely to contain test smells. The manually developed benchmarks, methodological advances, and empirical findings provide a more accurate foundation for future research on test evolution, regression testing, test obsolescence, and automated test repair.

Chapter 5

Conclusion and Future Work

5.1 Summary of Contributions

Software maintenance research often emphasizes production code, although test code also plays an important role in software maintenance. This thesis contributes to test-code maintenance research along two dimensions: technical debt in test code and test-method evolution. For technical debt, this thesis shows that SATD in test code is uncommon but diverse, includes test-specific categories not fully captured by production-code SATD, and remains difficult for existing detectors and large language models to identify reliably. For method evolution, the thesis shows that reliable method-history evidence requires carefully validated oracles, and that *HistoryFinder* provides the strongest overall support for reconstructing method histories among the evaluated research tools. Using validated test-method histories and method-level production-to-test mappings, the thesis further shows that many test methods evolve as frequently as, or more frequently than, the production methods they exercise, and that highly revised test methods are associated with more test smells. Together, these findings show that test code has maintenance concerns, detection challenges, and evolution patterns that are not fully captured by production-code assumptions. They also provide a foundation for future research on technical debt and evolution.

5.1.1 Characterizing Self-Admitted Technical Debt in Test Code

The SATD contribution characterizes how developers admit technical debt in test code by manually identifying relevant comments and classifying them into 12 categories, including four not captured by the source-code SATD taxonomies.

The findings show that this form of debt is uncommon but diverse. Some comments resemble known production-code debt categories, whereas others reflect test-specific concerns such as skipped tests and limited tests. Existing SATD detectors and large language models are also not yet reliable enough for detecting such debt in test comments. Overall, this contribution provides a test-specific taxonomy, a manually curated dataset, and empirical evidence on the limitations of current detection approaches.

5.1.2 Generating Reliable Method-History

The method-history work improves the reliability of method-level history generation. It shows that existing method-history oracles contain missing or incorrect revisions, which can affect conclusions about tool accuracy. To address this problem, the thesis constructs systematically validated oracles by combining the outputs of multiple tools with manual validation.

Using these oracles, the evaluation revisits existing method-history tools and introduces *HistoryFinder*. The results show that *HistoryFinder* provides the strongest overall balance of precision, recall, F1 score, and runtime among the evaluated research tools. This contribution strengthens the empirical foundation for method-level maintenance research and provides reusable support for studying how individual methods evolve.

5.1.3 Understanding Test Method Evolution

To assess whether existing method-history tools can support test-method evolution analysis, a manually validated oracle of test-method histories is constructed. The evaluation shows

that tools developed for production code method history generation can also generalize to test code, with *HistoryFinder* achieving the strongest overall performance. Method-level production-to-test mapping techniques are also evaluated, showing that no single approach dominates: some provide higher precision, whereas others provide broader recall.

The evolution analysis shows that test methods are not merely passive reflections of production-code change. Many test methods are revised as often as, or more often than, their corresponding production methods. Highly revised test methods also tend to contain more test smells, with the strongest association observed among small methods. These findings indicate that test methods have their own maintenance dynamics and should be studied as independently evolving software artifacts.

5.2 Future Work

5.2.1 Improving Test-Code SATD Detection

Future work should develop SATD detection approaches that are designed specifically for test comments and capable of learning test-specific expressions. Large language models also require more test-aware adaptation. It should investigate richer contextual inputs, such as surrounding test code, to better distinguish genuine SATD from ordinary explanatory comments.

Once test-code SATD can be detected more reliably, future work can also develop automated classification techniques based on the proposed taxonomy. Reliable detection and classification would enable larger-scale studies of the long-term maintenance impact of test-code SATD, such as whether affected tests are more likely to be changed or associated with later defects than comparable tests without SATD.

5.2.2 Extending Method-History Generation

Scalability can be improved by reducing the cost of method extraction during history construction. *HistoryFinder* currently relies on `JavaParser` to extract method-level information from `Git` revisions. Because this parsing is repeated across many revisions and methods, it can increase construction time for large repositories or when histories are generated for hundreds of thousands of methods. A more efficient alternative is low-level direct parsing, which would extract only the method information needed for tracking.

Method-history generation should also be extended beyond Java to evaluate whether current approaches generalize to other programming languages. This extension may require adjusting similarity thresholds for language-specific matching.

5.2.3 Building History-Aware Test Maintenance Tools

The findings of this thesis can inform practical tools that combine SATD comments, method histories, production-to-test mappings, and test smells. Such tools could help developers identify tests that contain self-admitted debt, are revised frequently, evolve differently from the production methods they exercise, or show signs of quality problems. History-aware tools could also support test maintenance tasks such as regression-test prioritization and obsolete test detection.

5.3 Concluding Remarks

This thesis shows that test code deserves direct attention in software maintenance research. By studying SATD in test code, building reliable method-history evidence, and analyzing method-level test-code evolution, the thesis demonstrates that test code contains maintenance concerns and evolution patterns that cannot be fully understood through production-code assumptions alone.

Overall, the thesis contributes to a broader view of software maintenance in which test

code receives direct attention as a maintained software artifact. Advancing this view can help researchers build stronger empirical models and help practitioners maintain test suites that remain reliable, useful, and trustworthy over time.

Bibliography

- [1] D. Kafura and G. Reddy, “The Use of Software Complexity Metrics in Software Maintenance,” *IEEE Transactions on Software Engineering*, vol. SE-13, no. 3, pp. 335–343, Mar. 1987.
- [2] J. Borstler and B. Paech, “The Role of Method Chains and Comments in Software Readability and Comprehension—An Experiment,” *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 886–898, Sep. 2016.
- [3] D. Sokolowski, P. Weisenburger, and G. Salvaneschi, “Change is the only constant: dynamic updates for workflows,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 350–362.
- [4] T. Zhang, Y. Lu, Y. Yu, X. Mao, Y. Zhang, and Y. Zhao, “How do developers adapt code snippets to their contexts? an empirical study of context-based code snippet adaptations,” *IEEE Transactions on Software Engineering*, vol. 50, no. 11, pp. 2712–2731, 2024.
- [5] D. Romano and M. Pinzger, “Using source code metrics to predict change-prone Java interfaces,” in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. Williamsburg, VA, USA: IEEE, Sep. 2011, pp. 303–312.
- [6] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, “An Exploratory Study of the Impact of Code Smells on Software Change-proneness,” in *2009 16th Working Conference on Reverse Engineering*. Lille, France: IEEE, Oct. 2009, pp. 75–84.

- [7] L. Pascarella, F. Palomba, and A. Bacchelli, “On the performance of method-level bug prediction: A negative result,” *Journal of Systems and Software*, vol. 161, p. 110493, Mar. 2020.
- [8] S. Chowdhury, G. Uddin, H. Hemmati, and R. Holmes, “Method-level Bug Prediction: Problems and Promises,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 4, pp. 1–31, May 2024.
- [9] S. Chowdhury, “The Good, the Bad, and the Monstrous: Predicting Highly Change-Prone Source Code Methods at Their Inception,” *ACM Transactions on Software Engineering and Methodology*, vol. 34, p. 3715006, Jan. 2025.
- [10] A. Potdar and E. Shihab, “An Exploratory Study on Self-Admitted Technical Debt,” in *2014 IEEE International Conference on Software Maintenance and Evolution*. Victoria, BC, Canada: IEEE, Sep. 2014, pp. 91–100.
- [11] E. D. S. Maldonado and E. Shihab, “Detecting and quantifying different types of self-admitted technical Debt,” in *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. Bremen, Germany: IEEE, Oct. 2015, pp. 9–15.
- [12] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, “Are test smells really harmful? An empirical study,” *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, Aug. 2015.
- [13] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, “An empirical investigation into the nature of test smells,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. Singapore Singapore: ACM, Aug. 2016, pp. 4–15.
- [14] S. Counsell, G. Destefanis, X. Liu, S. Eldh, A. Ermedahl, and K. Andersson, “Comparing Test and Production Code Quality in a Large Commercial Multicore System,” in

2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). Limassol, Cyprus: IEEE, Aug. 2016, pp. 86–91.

- [15] B. Van Bladel and S. Demeyer, “A Comparative Study of Code Clone Genealogies in Test Code and Production Code,” in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Taipa, Macao: IEEE, Mar. 2023, pp. 913–920.
- [16] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, “On the Relation of Test Smells to Software Code Quality,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Madrid: IEEE, Sep. 2018, pp. 1–12.
- [17] C. Seaman, R. L. Nord, P. Kruchten, and I. Ozkaya, “Technical Debt: Beyond Definition to Understanding Report on the Sixth International Workshop on Managing Technical Debt,” *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 2, pp. 32–34, Apr. 2015.
- [18] G. Bavota and B. Russo, “A large-scale empirical study on self-admitted technical debt,” in *Proceedings of the 13th International Conference on Mining Software Repositories*. Austin Texas: ACM, May 2016, pp. 315–326.
- [19] S. Wehaibi, E. Shihab, and L. Guerrouj, “Examining the Impact of Self-Admitted Technical Debt on Software Quality,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Suita: IEEE, Mar. 2016, pp. 179–188.
- [20] S. Chowdhury, H. Kidwai, and M. Asaduzzaman, “Evidence is All We Need: Do Self-Admitted Technical Debts Impact Method-Level Maintenance?” in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. Ottawa, ON, Canada: IEEE, Apr. 2025, pp. 813–825.

- [21] Z. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, “SATD detector: a text-mining-based self-admitted technical debt detection tool,” in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*. Gothenburg Sweden: ACM, May 2018, pp. 9–12.
- [22] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li, “Identifying self-admitted technical debt in open source projects using text mining,” *Empirical Software Engineering*, vol. 23, no. 1, pp. 418–451, Feb. 2018.
- [23] Y. Li, M. Soliman, and P. Avgeriou, “Automatic identification of self-admitted technical debt from four different sources,” *Empirical Software Engineering*, vol. 28, no. 3, p. 65, May 2023.
- [24] M. S. Sheikhaei, Y. Tian, S. Wang, and B. Xu, “An empirical study on the effectiveness of large language models for SATD identification and classification,” *Empirical Software Engineering*, vol. 29, no. 6, p. 159, Nov. 2024.
- [25] E. D. S. Maldonado, E. Shihab, and N. Tsantalis, “Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt,” *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1044–1062, Nov. 2017.
- [26] E. Gama, M. Paixao, M. I. Cortés, and L. Monteiro, “Towards Realistic SATD Identification through Machine Learning Models: Ongoing Research and Preliminary Results,” in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. Porto de Galinhas Brazil: ACM, Jul. 2024, pp. 675–676.
- [27] L. Rantala, M. Mäntylä, and D. Lo, “Prevalence, contents and automatic detection of KL-SATD,” in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2020, pp. 385–388.

- [28] K. Zhu, M. Yin, and Y. Li, “Detecting and Classifying Self-Admitted of Technical Debt with CNN-BiLSTM,” *Journal of Physics: Conference Series*, vol. 1955, no. 1, p. 012102, Jun. 2021.
- [29] D. OBrien, “Artifact for ESEC/FSE Paper: ”23 Shades of Self-Admitted Technical Debt: An Empirical Study on Machine Learning Software”,” Aug. 2022.
- [30] A. Bhatia, F. Khomh, B. Adams, and A. E. Hassan, “An Empirical Study of Self-Admitted Technical Debt in Machine Learning Software,” *ACM Transactions on Software Engineering and Methodology*, p. 3785001, Dec. 2025.
- [31] E. D. S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, “An Empirical Study on the Removal of Self-Admitted Technical Debt,” in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Shanghai: IEEE, Sep. 2017, pp. 238–248.
- [32] F. Zampetti, A. Serebrenik, and M. Di Penta, “Was self-admitted technical debt removal a real removal?: an in-depth perspective,” in *Proceedings of the 15th International Conference on Mining Software Repositories*. Gothenburg Sweden: ACM, May 2018, pp. 526–536.
- [33] —, “Automatically Learning Patterns for Self-Admitted Technical Debt Removal,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. London, ON, Canada: IEEE, Feb. 2020, pp. 355–366.
- [34] M. Iammarino, F. Zampetti, L. Aversano, and M. Di Penta, “An empirical study on the co-occurrence between refactoring actions and Self-Admitted Technical Debt removal,” *Journal of Systems and Software*, vol. 178, p. 110976, Aug. 2021.
- [35] J. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, “An exploratory study on the introduction and removal of different types of technical debt in deep learning frameworks,” *Empirical Software Engineering*, vol. 26, no. 2, p. 16, Mar. 2021.

- [36] A. Mastropaolo, M. Di Penta, and G. Bavota, “Towards Automatically Addressing Self-Admitted Technical Debt: How Far Are We?” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Luxembourg, Luxembourg: IEEE, Sep. 2023, pp. 585–597.
- [37] H. Azuma, S. Matsumoto, Y. Kamei, and S. Kusumoto, “An empirical study on self-admitted technical debt in Dockerfiles,” *Empirical Software Engineering*, vol. 27, no. 2, p. 49, Mar. 2022.
- [38] A. Pinna, M. I. Lunesu, S. Orrù, and R. Tonelli, “Investigation on Self-Admitted Technical Debt in Open-Source Blockchain Projects,” *Future Internet*, vol. 15, no. 7, p. 232, Jun. 2023.
- [39] G. Wilder, R. Miyamoto, S. Watson, R. Kazman, and A. Peruma, “An Exploratory Study on the Occurrence of Self-Admitted Technical Debt in Android Apps,” in *2023 ACM/IEEE International Conference on Technical Debt (TechDebt)*, May 2023, pp. 1–10.
- [40] S. Counsell and S. Swift, “Timing is Everything! A Test and Production Class View of Self-Admitted Technical Debt,” in *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. Gran Canaria, Spain: IEEE, Aug. 2022, pp. 314–317.
- [41] M. Beller, G. Gousios, A. Panichella, S. Proksch, S. Amann, and A. Zaidman, “Developer Testing in the IDE: Patterns, Beliefs, and Behavior,” *IEEE Transactions on Software Engineering*, vol. 45, no. 3, pp. 261–284, Mar. 2019.
- [42] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, “When, how, and why developers (do not) test in their IDEs,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. Bergamo Italy: ACM, Aug. 2015, pp. 179–190.

- [43] A. Zaidman, B. Van Rompaey, A. Van Deursen, and S. Demeyer, “Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining,” *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, Jun. 2011.
- [44] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, “Test Code Quality and Its Relation to Issue Handling Performance,” *IEEE Transactions on Software Engineering*, vol. 40, no. 11, pp. 1100–1125, Nov. 2014.
- [45] F. Grund, S. A. Chowdhury, N. C. Bradley, B. Hall, and R. Holmes, “CodeShovel: Constructing Method-Level Source Code Histories,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. Madrid, ES: IEEE, May 2021, pp. 1510–1522.
- [46] E. Duala-Ekoko and M. P. Robillard, “Tracking Code Clones in Evolving Software,” in *29th International Conference on Software Engineering (ICSE’07)*. Minneapolis, MN, USA: IEEE, May 2007, pp. 158–167.
- [47] Y. Kashiwabara, T. Ishio, H. Hata, and K. Inoue, “Method Verb Recommendation Using Association Rule Mining in a Set of Existing Projects,” *IEICE Transactions on Information and Systems*, vol. E98.D, no. 3, pp. 627–636, 2015.
- [48] X. Meng, B. P. Miller, W. R. Williams, and A. R. Bernat, “Mining Software Repositories for Accurate Authorship,” in *2013 IEEE International Conference on Software Maintenance*. Eindhoven, Netherlands: IEEE, Sep. 2013, pp. 250–259.
- [49] F. Rahman and P. Devanbu, “Ownership, experience and defects: a fine-grained study of authorship,” in *Proceedings of the 33rd International Conference on Software Engineering*. Waikiki, Honolulu HI USA: ACM, May 2011, pp. 491–500.
- [50] A. Zaidman, B. Van Rompaey, A. Van Deursen, and S. Demeyer, “Studying the co-evolution of production and test code in open source and industrial developer test

- processes through repository mining,” *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011.
- [51] S. Wang, M. Wen, Y. Liu, Y. Wang, and R. Wu, “Understanding and facilitating the co-evolution of production and test code,” in *2021 IEEE International conference on software analysis, evolution and reengineering (SANER)*, 2021, pp. 272–283.
- [52] W. Sun, M. Yan, Z. Liu, X. Xia, Y. Lei, and D. Lo, “Revisiting the Identification of the Co-evolution of Production and Test Code,” *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–37, Nov. 2023.
- [53] L. S. Pinto, S. Sinha, and A. Orso, “Understanding myths and realities of test-suite evolution,” in *Proceedings of the ACM SIGSOFT 20th international symposium on the foundations of software engineering*, 2012, pp. 1–11.
- [54] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, “On the relation of test smells to software code quality,” in *2018 IEEE International Conference on Software Maintenance and Evolution*, 2018, pp. 1–12.
- [55] D. Hao, T. Lan, H. Zhang, C. Guo, and L. Zhang, “Is this a bug or an obsolete test?” in *European Conference on Object-Oriented Programming*, 2013, pp. 602–628.
- [56] S. Bhatta, F. Kendemah, and A. K. Jha, “Understanding test deletion in java applications,” in *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. IEEE, 2025, pp. 408–420.
- [57] A. S. Yaraghi, D. Holden, N. Kahani, and L. Briand, “Automated test case repair using language models,” *IEEE Transactions on Software Engineering*, 2025.
- [58] H. Hata, O. Mizuno, and T. Kikuno, “Historage: fine-grained version control system for Java,” in *Proceedings of the 12th International Workshop on Principles of Soft-*

- ware Evolution and the 7th annual ERCIM Workshop on Software Evolution.* Szeged Hungary: ACM, Sep. 2011, pp. 96–100.
- [59] Y. Higo, S. Hayashi, and S. Kusumoto, “On tracking Java methods with Git mechanisms,” *Journal of Systems and Software*, vol. 165, p. 110571, Jul. 2020.
 - [60] M. Jodavi and N. Tsantalis, “Accurate method and variable tracking in commit history,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* Singapore Singapore: ACM, Nov. 2022, pp. 183–195.
 - [61] B. Van Rompaey and S. Demeyer, “Establishing traceability links between unit test cases and units under test,” in *2009 13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 209–218.
 - [62] R. White, J. Krinke, and R. Tan, “Establishing multilevel test-to-code traceability links,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, Jun. 2020, pp. 861–872.
 - [63] R. White and J. Krinke, “TCTracer: Establishing test-to-code traceability links using dynamic and static techniques,” *Empirical Software Engineering*, vol. 27, no. 3, p. 67, May 2022.
 - [64] W. Sun, Z. Guo, M. Yan, Z. Liu, Y. Lei, and H. Zhang, “Method-Level Test-to-Code Traceability Link Construction by Semantic Correlation Learning,” *IEEE Transactions on Software Engineering*, vol. 50, no. 10, pp. 2656–2676, Oct. 2024.
 - [65] S. Chowdhury, R. Holmes, A. Zaidman, and R. Kazman, “Revisiting the debate: Are code metrics useful for measuring maintenance effort?” *Empirical Software Engineering*, vol. 27, no. 6, p. 158, Aug. 2022.

- [66] Y. Zhou, B. Xu, and H. Leung, “On the ability of complexity metrics to predict fault-prone classes in object-oriented systems,” *Journal of Systems and Software*, vol. 83, no. 4, pp. 660–674, Apr. 2010.
- [67] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, “An Exploratory Study on the Relationship between Changes and Refactoring,” in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. Buenos Aires, Argentina: IEEE, May 2017, pp. 176–185.
- [68] D. Winkler, P. Urbanke, and R. Ramler, “Investigating the readability of test code,” *Empirical Software Engineering*, vol. 29, no. 2, p. 53, Mar. 2024.
- [69] A. Tahir, S. Counsell, and S. G. MacDonell, “An Empirical Study into the Relationship Between Class Features and Test Smells,” in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. Hamilton, New Zealand: IEEE, 2016, pp. 137–144.
- [70] A. Vahabzadeh, A. M. Fard, and A. Mesbah, “An empirical study of bugs in test code,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Bremen, Germany: IEEE, Sep. 2015, pp. 101–110.
- [71] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, “Investigating the impact of design debt on software quality,” in *Proceedings of the 2nd Workshop on Managing Technical Debt*. Waikiki, Honolulu HI USA: ACM, May 2011, pp. 17–23.
- [72] R. O. Spinola, A. Vetro, N. Zazworka, C. Seaman, and F. Shull, “Investigating technical debt folklore: Shedding some light on technical debt opinion,” in *2013 4th International Workshop on Managing Technical Debt (MTD)*. San Francisco, CA, USA: IEEE, May 2013, pp. 1–7.
- [73] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. B. Da Silva, A. L. M. Santos, and C. Siebra, “Tracking technical debt — An exploratory case

- study,” in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. Williamsburg, VA, USA: IEEE, Sep. 2011, pp. 528–531.
- [74] T. Xiao, Z. Zeng, D. Wang, H. Hata, S. McIntosh, and K. Matsumoto, “Quantifying and characterizing clones of self-admitted technical debt in build systems,” *Empirical Software Engineering*, vol. 29, no. 2, p. 54, Mar. 2024.
- [75] N. Cassee, F. Zampetti, N. Novielli, A. Serebrenik, and M. Di Penta, “Self-Admitted Technical Debt and comments’ polarity: an empirical study,” *Empirical Software Engineering*, vol. 27, no. 6, p. 139, Nov. 2022.
- [76] Z. Guo, S. Liu, J. Liu, Y. Li, L. Chen, H. Lu, and Y. Zhou, “How Far Have We Progressed in Identifying Self-admitted Technical Debts? A Comprehensive Empirical Study,” *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 4, pp. 1–56, Oct. 2021.
- [77] H. Gu, S. Zhang, Q. Huang, Z. Liao, J. Liu, and D. Lo, “Self-Admitted Technical Debts Identification: How Far Are We?” in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Rovaniemi, Finland: IEEE, Mar. 2024, pp. 804–815.
- [78] I. Sala, A. Tommasel, and F. Arcelli Fontana, “DebtHunter: A machine learning-based approach for detecting self-admitted technical debt,” in *Evaluation and Assessment in Software Engineering*. ACM, 2021, pp. 278–283.
- [79] L. Mayer, C. Heumann, and M. Aßenmacher, “Can OpenSource beat ChatGPT? – A Comparative Study of Large Language Models for Text-to-Code Generation,” 2024.
- [80] T. McCabe, “A Complexity Measure,” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

- [81] S. Chidamber and C. Kemerer, “A metrics suite for object oriented design,” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994.
- [82] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, “Decoupling level: a new metric for architectural maintenance complexity,” in *Proceedings of the 38th International Conference on Software Engineering*. Austin Texas: ACM, May 2016, pp. 499–510.
- [83] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, “An exploratory study of the impact of antipatterns on class change- and fault-proneness,” *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, Jun. 2012.
- [84] E. Giger, M. D’Ambros, M. Pinzger, and H. C. Gall, “Method-level bug prediction,” in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. Lund Sweden: ACM, Sep. 2012, pp. 171–180.
- [85] D. Posnett, A. Hindle, and P. Devanbu, “A simpler model of software readability,” in *Proceedings of the 8th Working Conference on Mining Software Repositories*. Waikiki, Honolulu HI USA: ACM, May 2011, pp. 73–82.
- [86] R. P. L. Buse and W. R. Weimer, “Learning a Metric for Code Readability,” *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, Jul. 2010.
- [87] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, “Automatically assessing code understandability: How far are we?” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Urbana, IL: IEEE, Oct. 2017, pp. 417–427.
- [88] P. Lambert, L. Ishitani, and L. Xavier, “On the Identification of Self-Admitted Technical Debt with Large Language Models,” in *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software (SBES 2024)*. Brasil: Sociedade Brasileira de Computação, Sep. 2024, pp. 651–657.

- [89] N. S. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spinola, “Towards an Ontology of Terms on Technical Debt,” in *2014 Sixth International Workshop on Managing Technical Debt*. Victoria, BC, Canada: IEEE, Sep. 2014, pp. 1–7.
- [90] F. Arcelli Fontana, J. Di Rocco, D. Di Ruscio, A. Di Salle, and P. T. Nguyen, “Binary and multi-class classification of Self-Admitted Technical Debt: How far can we go?” *Information and Software Technology*, vol. 187, p. 107862, Nov. 2025.
- [91] G. Fucci, N. Cassee, F. Zampetti, N. Novielli, A. Serebrenik, and M. Di Penta, “Waiting around or job half-done? Sentiment in self-admitted technical debt,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. Madrid, Spain: IEEE, May 2021, pp. 403–414.
- [92] V. K. Njeru, J. Kuria, and B. Kituku, “Hybrid BERT, CNN-BILSTM Model for Detecting Self-Admitted Technical Debt in Software Development,” in *2025 IEEE Conference on Computer Applications (ICCA)*, Mar. 2025, pp. 1–6.
- [93] J. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, “Is using deep learning frameworks free?: characterizing technical debt in deep learning frameworks,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society*. Seoul South Korea: ACM, Jun. 2020, pp. 1–10.
- [94] I. Nakamura, Y. Kashiwa, B. Lin, and H. Iida, “Understanding self-admitted technical debt in test code: An empirical study,” *ACM Transactions on Software Engineering and Methodology*, 2026.
- [95] S. Jahan, S. S. Rajput, T. Sharma, and M. M. Rahman, “Why attention fails: A taxonomy of faults in attention-based neural networks,” in *2026 IEEE/ACM International Conference on Software Engineering (ICSE)*, 2026.
- [96] B. Lin, C. Nagy, G. Bavota, A. Marcus, and M. Lanza, “On the Quality of Identifiers in

- Test Code,” in *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Cleveland, OH, USA: IEEE, Sep. 2019, pp. 204–215.
- [97] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, “The promises and perils of mining GitHub,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*. Hyderabad India: ACM, May 2014, pp. 92–101.
- [98] H. Borges and M. Tulio Valente, “What’s in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform,” *Journal of Systems and Software*, vol. 146, pp. 112–129, Dec. 2018.
- [99] J. Cohen, “A Coefficient of Agreement for Nominal Scales,” *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, Apr. 1960.
- [100] A. J. Viera, J. M. Garrett *et al.*, “Understanding interobserver agreement: the kappa statistic,” *Fam med*, vol. 37, no. 5, pp. 360–363, 2005.
- [101] M. S. Sheikhaei, Y. Tian, S. Wang, and B. Xu, “Understanding the Effectiveness of LLMs in Automated Self-Admitted Technical Debt Repayment,” 2025, version Number: 1.
- [102] X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, and J. Grundy, “Neural Network-based Detection of Self-Admitted Technical Debt: From Performance to Explainability,” *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 3, pp. 1–45, Jul. 2019.
- [103] S. Islam, A. Aowal, M. S. Uddin, and S. Chowdhury, “Historyfinder: Advancing method-level source code history generation with accurate oracles and enhanced algorithm,” in *48th IEEE/ACM International Conference on Software Engineering*, 2026.

- [104] M. N. I. Opu, S. Islam, S. Rouhani, and S. Chowdhury, “Understanding the issue types in open source blockchain-based software projects with the transformer-based bertopic,” *2025 IEEE International Conference on Collaborative Advances in Software and Computing*, 2025.
- [105] R. Maipradit, C. Treude, H. Hata, and K. Matsumoto, “Wait for it: identifying “On-Hold” self-admitted technical debt,” *Empirical Software Engineering*, vol. 25, no. 5, pp. 3770–3798, Sep. 2020.
- [106] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language models are few-shot learners,” in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS ’20, 2020.
- [107] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, “Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing,” *ACM Computing Surveys*, vol. 55, no. 9, pp. 1–35, Sep. 2023.
- [108] X. Amatriain, “Prompt Design and Engineering: Introduction and Advanced Methods,” 2024.
- [109] L. Giray, “Prompt Engineering with ChatGPT: A Guide for Academic Writers,” *Annals of Biomedical Engineering*, vol. 51, no. 12, pp. 2629–2633, Dec. 2023.
- [110] T. Hu and N. Collier, “Quantifying the Persona Effect in LLM Simulations,” 2024.
- [111] S. S. Dvivedi, V. Vijay, S. L. R. Pujari, S. Lodh, and D. Kumar, “A comparative analysis of large language models for code documentation generation,” in *Proceedings of the 1st ACM international conference on AI-powered software*, 2024, pp. 65–73.

- [112] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, “Swe-bench: Can language models resolve real-world github issues?” in *International Conference on Learning Representations*, vol. 2024, 2024, pp. 54 107–54 157.
- [113] S. S. Batista, B. Branco, O. Castro, and G. Avelino, “Code on demand: A comparative analysis of the efficiency understandability and self-correction capability of copilot chatgpt and gemini,” in *Proceedings of the XXIII Brazilian Symposium on Software Quality*, 2024, pp. 351–361.
- [114] M. Bruni, F. Gabrielli, M. Ghafari, and M. Kropp, “Benchmarking prompt engineering techniques for secure code generation with gpt models,” in *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*. IEEE, 2025, pp. 93–103.
- [115] I. R. d. S. Simões and E. Venson, “Evaluating source code quality with large language models: a comparative study,” in *Proceedings of the XXIII Brazilian Symposium on Software Quality*, 2024, pp. 103–113.
- [116] E. Mashhadi, S. Chowdhury, S. Modaberi, H. Hemmati, and G. Uddin, “An empirical study on bug severity estimation using source code metrics and static analysis,” *Journal of Systems and Software*, vol. 217, p. 112179, Nov. 2024.
- [117] Y. Li, T. Zhang, R. Widyasari, Y. N. Tun, H. H. Nguyen, T. Bui, I. C. Irsan, Y. Cheng, X. Lan, H. W. Ang, F. Liauw, M. Weyssow, H. J. Kang, E. L. Ouh, L. K. Shar, and D. Lo, “CleanVul: Automatic Function-Level Vulnerability Detection in Code Commits Using LLM Heuristics,” 2024.
- [118] K. Hassan, S. Moradi, S. Chowdhury, and S. Rouhani, “Versioned Analysis of Software Quality Indicators and Self-admitted Technical Debt in Ethereum Smart Contracts with Ethstractor,” in *2024 IEEE International Conference on Blockchain (Blockchain)*. Copenhagen, Denmark: IEEE, Aug. 2024, pp. 512–519.

- [119] S. I. Ahmad, S. Chowdhury, and R. Holmes, “Impact of methodological choices on the analysis of code metrics and maintenance,” *Journal of Systems and Software*, vol. 220, p. 112263, Feb. 2025.
- [120] E.-M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, “A Method for Assessing Class Change Proneness,” in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. Karlskrona Sweden: ACM, Jun. 2017, pp. 186–195.
- [121] L. Di Grazia and M. Pradel, “Code Search: A Survey of Techniques for Finding Code,” *ACM Computing Surveys*, vol. 55, no. 11, pp. 1–31, Nov. 2023.
- [122] M. N. I. Opu, S. Wang, and S. Chowdhury, “Llm-based detection of tangled code changes for higher-quality method-level bug datasets,” 2025.
- [123] A. Güneş Koru and H. Liu, “Identifying and characterizing change-prone classes in two large-scale open-source products,” *Journal of Systems and Software*, vol. 80, no. 1, pp. 63–73, Jan. 2007.
- [124] R. Abbas, F. A. Albalooshi, and M. Hammad, “Software Change Proneness Prediction Using Machine Learning,” in *2020 International Conference on Innovation and Intelligence for Informatics, Computing and Technologies (3ICT)*. Sakheer, Bahrain: IEEE, Dec. 2020, pp. 1–7.
- [125] S. A. Chowdhury, G. Uddin, and R. Holmes, “An empirical study on maintainable method size in Java,” in *Proceedings of the 19th International Conference on Mining Software Repositories*. Pittsburgh Pennsylvania: ACM, May 2022, pp. 252–264.
- [126] R. Aleithan, “Explainable Just-In-Time Bug Prediction: Are We There Yet?” in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. Madrid, ES: IEEE, May 2021, pp. 129–131.

- [127] H. Alsolai, M. Roper, and D. Nassar, “Predicting Software Maintainability in Object-Oriented Systems Using Ensemble Techniques,” in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Madrid, Spain: IEEE, Sep. 2018, pp. 716–721.
- [128] V. Basili, L. Briand, and W. Melo, “A validation of object-oriented design metrics as quality indicators,” *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, Oct. 1996.
- [129] Y. Gil and G. Lalouche, “On the correlation between size and metric validity,” *Empirical Software Engineering*, vol. 22, no. 5, pp. 2585–2611, Oct. 2017.
- [130] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting Defects for Eclipse,” in *Third International Workshop on Predictor Models in Software Engineering (PROMISE’07: ICSE Workshops 2007)*. Minneapolis, MN, USA: IEEE, May 2007, pp. 9–9.
- [131] Y. Kamei and E. Shihab, “Defect Prediction: Accomplishments and Future Challenges,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5. Osaka, Japan: IEEE, Mar. 2016, pp. 33–45.
- [132] R. Mo, S. Wei, Q. Feng, and Z. Li, “An exploratory study of bug prediction at the method level,” *Information and Software Technology*, vol. 144, p. 106794, Apr. 2022.
- [133] R. Ferenc, P. Gyimesi, G. Gyimesi, Z. Tóth, and T. Gyimóthy, “An automatically created novel bug dataset and its validation in bug prediction,” *Journal of Systems and Software*, vol. 169, p. 110691, Nov. 2020.
- [134] T. Shippey, T. Hall, S. Counsell, and D. Bowes, “So You Need More Method Level Datasets for Your Software Defect Prediction?: Voilà!” in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. Ciudad Real Spain: ACM, Sep. 2016, pp. 1–6.

- [135] H. Hata, O. Mizuno, and T. Kikuno, “Bug prediction based on fine-grained module histories,” in *2012 34th International Conference on Software Engineering (ICSE)*. Zurich, Switzerland: IEEE, Jun. 2012, pp. 200–210.
- [136] T. Menzies, J. Greenwald, and A. Frank, “Data Mining Static Code Attributes to Learn Defect Predictors,” *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2–13, Jan. 2007.
- [137] S. Kim, K. Pan, and E. Whitehead, “When functions change their names: automatic detection of origin relationships,” in *12th Working Conference on Reverse Engineering (WCRE’05)*. Pittsburgh, PA: IEEE, Nov. 2005, pp. 10 pp.–152.
- [138] Q. Tu and M. Godfrey, “An integrated approach for studying architectural evolution,” in *Proceedings 10th International Workshop on Program Comprehension*. Paris, France: IEEE, Jun. 2002, pp. 127–136.
- [139] T. Zimmermann, “Fine-grained processing of CVS archives with APFEL,” in *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange - eclipse ’06*. Portland, Oregon: ACM Press, 2006, pp. 16–20.
- [140] A. E. Hassan and R. Holt, “C-rex: An evolutionary code extractor for c,” 09 2004.
- [141] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, “ClDiff: generating concise linked code differences,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Montpellier France: ACM, Sep. 2018, pp. 679–690.
- [142] J.-R. Falleri and M. Martinez, “Fine-grained, accurate and scalable source differencing,” in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Lisbon Portugal: ACM, Apr. 2024, pp. 1–12.

- [143] N. Tsantalis, A. Ketkar, and D. Dig, “RefactoringMiner 2.0,” *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, Mar. 2022.
- [144] W. E. Winkler, “String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage,” in *Proceedings of the Section on Survey Research*. Washington, DC: American Statistical Association, 1990, pp. 354–359.
- [145] M. S. Charikar, “Similarity estimation techniques from rounding algorithms,” in *Proceedings of the thirty-fourth annual ACM symposium on Theory of computing*. Montreal Quebec Canada: ACM, May 2002, pp. 380–388.
- [146] K. Geisshirt, E. Zattin, A. Olsson, and R. Voss, *Git version control cookbook: leverage version control to transform your development workflow and boost productivity, second edition*, 2nd ed. Birmingham: Packt Publishing, 2018.
- [147] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. Cambridge, Massachusetts London: The MIT Press, 2022.
- [148] F. Wilcoxon, “Individual Comparisons by Ranking Methods,” *Biometrics Bulletin*, vol. 1, no. 6, p. 80, Dec. 1945.
- [149] C. Vassallo, G. Grano, F. Palomba, H. C. Gall, and A. Bacchelli, “A large-scale empirical exploration on refactoring activities in open source software projects,” *Science of Computer Programming*, vol. 180, pp. 1–15, 2019.
- [150] E. Kalliamvakou, D. Damian, K. Blincoe, L. Singer, and D. M. German, “Open source-style collaborative development practices in commercial projects using github,” in *2015 IEEE/ACM 37th IEEE international conference on software engineering*, vol. 1, 2015, pp. 574–585.
- [151] A. Martini, J. Bosch, and M. Chaudron, “Investigating architectural technical debt

- accumulation and refactoring over time: A multiple-case study,” *Information and Software Technology*, vol. 67, pp. 237–253, 2015.
- [152] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, and A. Ampatzoglou, “How do developers fix issues and pay back technical debt in the apache ecosystem?” in *2018 IEEE 25th International Conference on software analysis, evolution and reengineering (SANER)*, 2018, pp. 153–163.
- [153] A. A. B. Baqais and M. Alshayeb, “Automatic software refactoring: a systematic literature review,” *Software Quality Journal*, vol. 28, no. 2, pp. 459–502, 2020.
- [154] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, “Predicting defective lines using a model-agnostic technique,” *IEEE Transactions on Software Engineering*, vol. 48, no. 05, pp. 1480–1496, may 2022.
- [155] Z. Wang, W. Tong, P. Li, G. Ye, H. Chen, X. Gong, and Z. Tang, “Bugpre: an intelligent software version-to-version bug prediction system using graph convolutional neural networks,” *Complex & Intelligent Systems*, pp. 1–21, 2022.
- [156] L. Pascarella, F. Palomba, and A. Bacchelli, “On the performance of method-level bug prediction: A negative result,” *Journal of Systems and Software*, vol. 161, 2020.
- [157] M. S. Rahman and C. K. Roy, “On the relationships between stability and bug-proneness of code clones: An empirical study,” in *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2017, pp. 131–140.
- [158] E.-M. Arvanitou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, “A method for assessing class change proneness,” in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, 2017, pp. 186–195.
- [159] S. Chowdhury, G. Uddin, and R. Holmes, “An empirical study on maintainable method

- size in java,” in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2022, pp. 252–264.
- [160] S. Chowdhury, R. Holmes, A. Zaidman, and R. Kazman, “Revisiting the debate: Are code metrics useful for measuring maintenance effort?” *Empirical Software Engineering (EMSE)*, vol. 27, no. 6, 2022.
- [161] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, “An industrial study on the risk of software changes,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012.
- [162] A. Mastropaolo, M. Di Penta, and G. Bavota, “Towards automatically addressing self-admitted technical debt: How far are we?” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 585–597.
- [163] F. Grund, S. Chowdhury, N. C. Bradley, B. Hall, and R. Holmes, “Codeshovel: Constructing method-level source code histories,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2021, pp. 1510–1522.
- [164] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig, “Is it dangerous to use version control histories to study source code evolution?” in *European Conference on Object-Oriented Programming*, 2012, pp. 79–103.
- [165] R. Funaki, S. Hayashi, and M. Saeki, “The impact of systematic edits in history slicing,” in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2019, pp. 555–559.
- [166] Y. Li, C. Zhu, J. Rubin, and M. Chechik, “Semantic slicing of software version histories,” *Transactions on Software Engineering (TSE)*, vol. 44, no. 2, pp. 182–201, 2018.
- [167] R. Mo, S. Wei, Q. Feng, and Z. Li, “An exploratory study of bug prediction at the method level,” *Inf. Softw. Technol.*, vol. 144, no. C, apr 2022.

- [168] G. Canfora, L. Cerulo, and M. Di Penta, “Identifying changed source code lines from version repositories,” in *Proceedings of the Workshop on Mining Software Repositories (MSR)*, 2007, pp. 14–21.
- [169] A. Chen, E. Chou, J. Wong, A. Y. Yao, Qing Zhang, Shao Zhang, and A. Michail, “CVSSearch: Searching through source code using CVS comments,” in *Proceedings of the International Conference on Software Maintenance (ICSM)*, 2001, pp. 364–373.
- [170] D. Steidl, B. Hummel, and E. Juergens, “Incremental origin analysis of source code files,” in *Proceedings Working Conference on Mining Software Repositories (MSR)*, 2014, pp. 42—51.
- [171] F. Servant and J. A. Jones, “Fuzzy fine-grained code-history analysis,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2017, pp. 746–757.
- [172] Y. Higo, S. Hayashi, and S. Kusumoto, “On tracking java methods with git mechanisms,” *Journal of Systems and Software*, vol. 165, p. 110571, 2020.
- [173] S. Islam, A. Aowal, M. S. Uddin, and S. Chowdhury, “HistoryFinder: Advancing Method-Level Source Code History Generation with Accurate Oracles and Enhanced Algorithm,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, Apr. 2026.
- [174] S. Chowdhury, G. Uddin, H. Hemmati, and R. Holmes, “Method-level bug prediction: Problems and promises,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 4, pp. 1–31, 2024.
- [175] S. Chowdhury, “The good, the bad, and the monstrous: Predicting highly change-prone source code methods at their inception,” *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 7, pp. 1–29, 2025.

- [176] N. Tsantalis, A. Ketkar, and D. Dig, “Refactoringminer 2.0,” *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2020.
- [177] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, “An experimental investigation on the innate relationship between quality and refactoring,” *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015.
- [178] X. Hu, Z. Liu, X. Xia, Z. Liu, T. Xu, and X. Yang, “Identify and update test cases when production code changes: A transformer-based approach,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1111–1122.
- [179] H. Hata, O. Mizuno, and T. Kikuno, “Hstorage: Fine-grained version control system for java,” in *Proc. International Workshop on Principles of Software Evolution and ERCIM Workshop on Software Evolution*, 2011, pp. 96–100.
- [180] B. Van Bladel and S. Demeyer, “A comparative study of test code clones and production code clones,” *Journal of Systems and Software*, vol. 176, p. 110940, 2021.
- [181] K. Horikawa, Y. Kashiwa, B. Lin, K. Fujiwara, and H. Iida, “How does test code differ from production code in terms of refactoring? an empirical study,” in *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2025, pp. 779–784.
- [182] Y. Liu, Z. Chen, L. Chen, Y. Li, X. Li, and W. Song, “COTE: Predicting Code-to-Test Co-Evolution by Integrating Link Analysis and Pre-Trained Language Model Techniques,” *IEEE Transactions on Software Engineering*, vol. 51, no. 8, pp. 2232–2253, Aug. 2025.
- [183] J. A. Jones, M. J. Harrold, and J. Stasko, “Visualization of test information to assist fault localization,” in *Proceedings of the 24th international conference on Software engineering*, 2002, pp. 467–477.

- [184] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, and Y. Le Traon, “Bench4bl: reproducibility study on the performance of ir-based bug localization,” in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 61–72.
- [185] E. Friesen, S. Morton-Salmon, M. N. I. Opu, S. Islam, and S. Chowdhury, “The Repeat Offenders: Characterizing and Predicting Extremely Bug-Prone Source Methods,” 2025.
- [186] M. Tufano, S. K. Deng, N. Sundaresan, and A. Svyatkovskiy, “Methods2test: A dataset of focal methods mapped to test cases,” in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 299–303.
- [187] I. Heitlager, T. Kuipers, and J. Visser, “A practical model for measuring maintainability,” in *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, 2007, p. 30–39.
- [188] F. Arcelli Fontana, V. Ferme, M. Zanoni, and A. Yamashita, “Automatic metric thresholds derivation for code smell detection,” in *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*, 2015, pp. 44–53.
- [189] S. Levin and A. Yehudai, “The co-evolution of test maintenance and code maintenance through the lens of fine-grained semantic changes,” in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 35–46.
- [190] C. Marsavina, D. Romano, and A. Zaidman, “Studying fine-grained co-evolution patterns of production and test code,” in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 195–204.
- [191] W. J. Dzidek, E. Arisholm, and L. C. Briand, “A realistic empirical evaluation of the costs and benefits of uml in software maintenance,” *IEEE Transactions on software engineering*, vol. 34, no. 3, pp. 407–432, 2008.

- [192] F. Zhang, A. Mockus, Y. Zou, F. Khomh, and A. E. Hassan, “How does context affect the distribution of software maintainability metrics?” in *IEEE International Conference on Software Maintenance*, 2013, pp. 350–359.
- [193] L. Inozemtseva and R. Holmes, “Coverage is not strongly correlated with test suite effectiveness,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2014, pp. 435–445.
- [194] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, “Are test smells really harmful? an empirical study,” *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, 2015.
- [195] A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn, “Test smells 20 years later: detectability, validity, and reliability,” *Empirical Software Engineering*, vol. 27, no. 7, p. 170, 2022.
- [196] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, “An empirical investigation into the nature of test smells,” in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 4–15.
- [197] T. Virgínio, L. Martins, L. Rocha, R. Santana, A. Cruz, H. Costa, and I. Machado, “Jnose: Java test smell detector,” in *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, 2020, pp. 564–569.
- [198] V. Garousi and B. Küçük, “Smells in software test code: A survey of knowledge in industry and academia,” *Journal of systems and software*, vol. 138, pp. 52–81, 2018.
- [199] R. A. Fisher, “On the interpretation of χ^2 from contingency tables, and the calculation of p,” *Journal of the royal statistical society*, vol. 85, no. 1, pp. 87–94, 1922.

- [200] Y. Benjamini and Y. Hochberg, “Controlling the false discovery rate: a practical and powerful approach to multiple testing,” *Journal of the Royal statistical society: series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.
- [201] N. Mantel and W. Haenszel, “Statistical aspects of the analysis of data from retrospective studies of disease,” *Journal of the national cancer institute*, vol. 22, no. 4, pp. 719–748, 1959.
- [202] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai, “The confounding effect of class size on the validity of object-oriented metrics,” *IEEE Transactions on Software Engineering*, vol. 27, no. 7, pp. 630–650, 2001.
- [203] Y. Gil and G. Lalouche, “On the correlation between size and metric validity,” *Empirical Software Engineering*, vol. 22, no. 5, p. 2585–2611, Oct. 2017.
- [204] D. I. K. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, “Quantifying the effect of code smells on maintenance effort,” *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2013.
- [205] D. Landman, A. Serebrenik, and J. Vinju, “Empirical analysis of the relationship between cc and sloc in a large corpus of java methods,” in *IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 221–230.
- [206] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, “An exploratory study of the impact of antipatterns on class change- and fault-proneness,” *Empirical software engineering : an international journal*, vol. 17, no. 3, pp. 243–275, 2012.

Contributions of Authors

This thesis follows a grouped manuscript format and contains three manuscripts. The nature and extent of the candidate’s contribution and the contributions of the co-authors are described below.

Manuscript 1 (Chapter 2): A First Look at the Self-Admitted Technical Debt in Test Code: Taxonomy and Detection

Authors: Shahidul Islam, Md Nahidul Islam Opu, Shaowei Wang, Shaiful Chowdhury.

Status: Under revision at The Journal of Systems & Software, 2026.

Candidate’s contribution: The candidate was responsible for the implementation, data collection, analysis, and writing. Contributed to the study design, response to reviewers, and manual SATD identification and classification.

Co-authors’ contributions: Shaiful Chowdhury and Shaowei Wang provided supervision, conceptual guidance, and feedback on the study. Md Nahidul Islam Opu and Shaiful Chowdhury contributed to the initial manual identification and classification of a subset of SATD comments.

Manuscript 2 (Chapter 3): HistoryFinder: Advancing Method-Level Source Code History Generation with Accurate Oracles and Enhanced Algorithm

Authors: Shahidul Islam, Ashik Aowal, Md Sharif Uddin, Shaiful Chowdhury.

Status: Accepted for publication in the proceedings of the 48th IEEE/ACM International Conference on Software Engineering (ICSE), 2026.

Candidate’s contribution: The candidate was responsible for the problem formulation, methodology and study design, implementation, data collection, oracle construction and validation, analysis, writing, revision, and response to reviewers.

Co-authors’ contributions: Shaiful Chowdhury provided supervision, conceptual guidance, methodology feedback, manuscript feedback, and response to reviewers. Ashik Aowal supported manual validation of method-history oracles. Md Sharif Uddin contributed to algorithmic design.

Manuscript 3 (Chapter 4): Understanding Method-Level Test Code Evolution

Authors: Shahidul Islam, Anusha Waseem, Md Nahidul Islam Opu, Shaiful Chowdhury.

Status: Under review for the 49th IEEE/ACM International Conference on Software Engineering (ICSE), Dublin, Ireland, 2027.

Candidate’s contribution: The candidate was responsible for the problem formulation, methodology and study design, implementation, data collection, production-to-test mapping setup, analysis, and writing. Contributed to the manual validation and annotation of test-method histories and production-to-test links.

Co-authors’ contributions: Shaiful Chowdhury provided supervision, conceptual guid-

ance, methodology feedback, and manuscript writing. Anusha Waseem supported oracle labeling. Md Nahidul Islam Opu supported validation of the implementation.