

The Limits and Promise of Software-Metric-Based Prediction: From Method-Level Bugs to App Popularity

by

Md Nahidul Islam Opu

A thesis submitted to
The Faculty of Graduate and Postdoctoral Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
August 2026

© Copyright 2026 by Md Nahidul Islam Opu

Thesis advisor

Author

Shaiful Chowdhury

Md Nahidul Islam Opu

The Limits and Promise of Software-Metric-Based Prediction: From Method-Level Bugs to App Popularity

Abstract

Software metrics have long been used to build predictive models of software quality, yet their effectiveness remains debated across prediction tasks. This thesis examines both the promise and the limits of software-metric-based prediction through three complementary studies spanning method-level bug prediction and mobile app popularity prediction. The first study addresses a fundamental source of noise in bug prediction datasets by introducing a Large Language Model (LLM)-based approach for detecting tangled code changes at the method level. By combining commit messages with method-level code diffs, the approach produces cleaner bug labels. Using these datasets, the second study revisits method-level bug prediction across 100 open-source projects, integrating traditional software metrics with source-code embeddings and evaluating realistic inception and evolution scenarios. Although cleaner labels and richer representations improve validation performance, limited cross-project generalization and severe class imbalance constrain test performance, highlighting persistent challenges in fine-grained bug prediction. The third study investigates whether internal source-code metrics can predict Android app popularity before release. While metrics cannot accurately predict continuous popularity measures, they support meaning-

ful binary classification. Collectively, these studies demonstrate that software metrics are valuable when paired with high-quality data, appropriate representations, and well-defined prediction objectives, but remain limited for fine-grained and exact-value prediction tasks.

Preface

This thesis is mainly based on three research papers, for which I was the main contributor under the supervision of Dr. Shaiful Chowdhury. I designed the methodologies, collected and prepared the datasets, implemented the proposed approaches, conducted the experiments, analyzed and evaluated the results, and wrote the manuscripts. My collaborators contributed through research discussions, methodological feedback, domain expertise, and manuscript review.

Chapter 2 of this thesis has been accepted as:

- Md Nahidul Islam Opu, Shaowei Wang, and Shaiful Chowdhury, “LLM-Based Detection of Tangled Code Changes for Higher-Quality Method-Level Bug Datasets,” accepted at the 23rd International Conference on Mining Software Repositories (MSR 2026), Technical Track, Rio de Janeiro, Brazil, 2026.

Chapter 3 of this thesis is based on the following paper, which is currently under review at a conference:

- Md Nahidul Islam Opu, Md. Masud Mazumder, Mohammad Asaduzzaman, and Shaiful Chowdhury, “Method-Level Bug Prediction: How Far Are We?”, under review.

Chapter 4 of this thesis is based on the following paper, which is currently under review at a journal:

- Md Nahidul Islam Opu, Fatima Islam Mouri, Rick Kazman, Yuanfang Cai, and Shaiful Chowdhury, “Moderately Mighty: To What Extent Can Internal Software Metrics Predict App Popularity at Launch?”, under review.

Below is the list of my other publications (including papers I co-authored) that I published or completed during my MSc degree, but are not included in this thesis:

- Md Nahidul Islam Opu, Md Shahidul Islam, Sara Rouhani, and Shaiful Chowdhury, “Understanding the Issue Types in Open Source Blockchain-based Software Projects with the Transformer-based BERTopic,” in *Proceedings of the 35th IEEE International Conference on Collaborative Advances in Software and Computing (CASCON 2025)*, pp. 454–463, November 10–13, 2025, Toronto, Canada.
- Md Nahidul Islam Opu, Md Shahidul Islam, Muhammad Asaduzzaman, and Shaiful Chowdhury, “How Do Agentic AI Systems Address Performance Optimizations? A BERTopic-Based Analysis of Pull Requests,” accepted at the 23rd International Conference on Mining Software Repositories (MSR 2026), Challenge Track, Rio de Janeiro, Brazil, 2026.
- Mumtahina Ahmed, Md Nahidul Islam Opu, Chanchal Roy, Sujana Islam Suhi, and Shaiful Chowdhury, “Exploring Challenges in Test Mocking: Developer Questions and Insights from StackOverflow,” *Journal of Systems and Software*, vol. 235, Article 112748, 2026.
- Shahidul Islam, Md Nahidul Islam Opu, Shaowei Wang, and Shaiful Chowdhury, “A First Look at the Self-Admitted Technical Debt in Test Code: Taxonomy and Detection,” under review.
- Ethan Friesen, Sasha Morton-Salmon, Md Nahidul Islam Opu, Shahidul Islam, and Shaiful Chowdhury, “The Repeat Offenders: Characterizing and Predicting

Extremely Bug-Prone Source Methods,” under review.

- Tanjum Motin Mitul, Md. Masud Mazumder, Md Nahidul Islam Opu, and Shaiful Chowdhury, “How Do Agentic AI Systems Deal With Software Energy Concerns? A Pull Request-Based Study,” accepted at the 23rd International Conference on Mining Software Repositories (MSR 2026), Challenge Track, Rio de Janeiro, Brazil, 2026.
- Hisham Kidwai, Danika Passler Bates, Sujana Islam Suhi, Md Nahidul Islam Opu, James E. Young, and Shaiful Chowdhury, “Robotics Meets Software Engineering: A First Look at the Robotics Discussions on StackOverflow,” *Journal of Software: Evolution and Process*, 2026.

*To my mother and father,
whose love, sacrifices, and support made this journey possible.*

Acknowledgments

First and foremost, I thank Almighty Allah for His countless blessings and for giving me the strength and perseverance to complete this thesis.

I would like to express my sincere gratitude to my supervisor, Professor Shaiful Chowdhury, for his instrumental support throughout this journey. His guidance, encouragement, kindness, and unwavering support have been invaluable to both my research and personal growth. I am also grateful to my committee members for their insightful feedback and valuable suggestions.

I sincerely thank my lab mates for their support, encouragement, and companionship, particularly during challenging times. Finally, I am deeply grateful to my family for their continuous love, patience, and unwavering support.

I am also grateful to have received the following recognitions:

- International Graduate Students Entrance Scholarship, University of Manitoba, Canada.
- Faculty of Graduate Studies Research Completion Scholarship, University of Manitoba, Canada.

Contents

1	Introduction	1
1.1	Software-Metric-Based Prediction	2
1.2	Motivation	4
1.2.1	Method-Level Bug Prediction	5
1.2.2	App Popularity Prediction	6
1.3	Contributions	7
1.3.1	A Realistic Evaluation of Method-Level Bug Prediction	7
	LLM-Based Detection of Tangled Code Changes for Higher Quality Method-Level Bug Datasets	8
	Method-Level Bug Prediction: How Far Are We?	8
1.3.2	A Launch-Time Evaluation of App Popularity Prediction Using Internal Metrics	9
1.4	Thesis Organization	9
I	Method-Level Bug Prediction	11
2	LLM-Based Detection of Tangled Code Changes for Higher-Quality Method-Level Bug Datasets	13
2.1	Introduction	14
2.2	Related Work	18
2.2.1	Tangled Changes	19
2.2.2	LLMs for Code Analysis & Classification	21
2.3	Methodology	22
2.3.1	Dataset	22
2.3.2	Prompt Preparation	26
2.3.3	Selecting and Querying LLMs	28
2.3.4	Evaluation Metrics	30
2.3.5	Experimental Setup	30
2.4	Approach, Analysis & Results	31

2.4.1	RQ1: Performance of zero-shot LLMs using code diff and commit message	31
2.4.2	RQ2: Performance of different prompting techniques	34
	Case analysis: an example of the effectiveness of chain-of-thought approach in a semantically ambiguous commit	37
2.4.3	RQ3: Performance of embedding-based machine learning models	39
2.4.4	RQ4: Potential impact of LLM-based untangling on future method-level bug prediction models	43
2.5	Threats to Validity	48
2.6	Conclusion and Future Work	49
3	Method-Level Bug Prediction: How Far Are We?	51
3.1	Introduction	52
3.2	Related Work and Motivation	55
3.2.1	Method-Level Bug Prediction and Prediction Granularity	56
3.2.2	Bug Labeling: Keyword Heuristics and Tangled Changes	57
3.2.3	Code, Change, and Developer Metrics	57
3.2.4	Learned Code Representations	58
3.2.5	Realistic and Temporal Evaluation	59
3.3	Methodology	60
3.3.1	Data Collection	60
	Project Selection	60
	History Collection	62
	Dataset Creation	62
3.3.2	Feature Extraction	64
	Tabular Features	65
	Code Embeddings	66
3.3.3	Classification Models	67
	Traditional Machine Learning (ML) Models	68
	Fusion Models	68
3.3.4	Evaluation	69
	Dataset Splits	70
	Evaluation Metrics	70
	Statistical Tests	71
	Evaluation Setup	72
3.4	Results	72
3.4.1	RQ1: Predictability at Inception	72
	Baseline on the Noisy dataset	73
	Effect of untangling - the Less-Noisy dataset	74
	Adding code semantics - the fusion model	76
3.4.2	RQ2: Predictability Through Evolution	80
	Baseline on the Noisy dataset	80

	Effect of untangling: the Less-Noisy dataset	82
	Adding code semantics: the fusion model	83
3.5	Discussion	88
3.5.1	Implications	88
	Cleaning labels is worth the effort	88
	Class imbalance, not project size, is the dominant limit	89
	Project composition drives the headline number	90
	Within-project signal beats cross-project	90
3.5.2	Future Work	90
3.5.3	Threats to Validity	91
3.6	Conclusion	92

II App Popularity Prediction 93

4	Moderately Mighty: To What Extent Can Internal Software Metrics Predict App Popularity at Launch?	95
4.1	Introduction	96
4.2	Related Work and Motivation	101
4.2.1	App Popularity Analysis	101
4.2.2	Code Metrics	104
4.2.3	Code Smells	106
4.2.4	Other Metadata	107
4.3	Methodology	109
4.3.1	App Collection	109
4.3.2	Popularity Indicators	111
4.3.3	Feature Extraction	112
4.3.4	Feature Selection	129
4.3.5	Learning Algorithms and Evaluation Metrics	133
4.4	Approach, Analysis, and Results	135
4.4.1	RQ1: To what extent can internal software metrics predict an app's user rating?	135
	Regression: Predicting the Actual Ratings	135
	Classification: Predicting Popularity Based on Ratings.	142
4.4.2	RQ2: To what extent can internal software metrics predict an app's number of downloads?	147
	Regression: Predicting the Actual <i>DownloadsPerYear</i>	147
	Classification: Predicting Popularity Based on <i>DownloadsPerYear</i>	151
4.5	Discussion	154
4.5.1	Threats to Validity	156
4.6	Conclusion	157

5	Conclusions and Future Work	159
5.1	Summary of the Contributions	159
5.1.1	Removing Tangled Noise	160
5.1.2	How Far Are We from Method-Level Bug Prediction?	160
5.1.3	Predicting App Popularity at Launch	161
5.2	Future Work	162
5.2.1	A Generalizable Method-Level Bug Prediction Model	162
5.2.2	Improving App Popularity Prediction	163
	Bibliography	164

List of Figures

2.1	The methodology for <i>RQ1</i> and <i>RQ2</i> consists of three parts: (a) dataset, (b) prompt preparation, and (c) LLM Query. The prompt setup for few-shot + chain-of-thought prompting consists of four instruction parts: (i) persona, (ii) task description, (iii) behavioral guidance, and (iv) output formatting. For Diff-only detection of <i>RQ1</i> , references to the commit message are excluded from everywhere. Other experiments use the full instructions, with the output formatting modified to require a single-word answer in non-chain-of-thought setups. Few-shot examples are included only in few-shot experiments.	23
2.2	Commit message and two methods modified in the same commit: the message indicates a bug fix, but only the method on the right contains the bug-fix change, while the method on the left is a non-bug-related tangled change. This was used as few-shot examples in our prompts. .	27
2.3	The methodology for <i>RQ3</i> has two main parts: (i) embedding generation and (ii) building ML models.	40
2.4	Distribution of code metrics for <i>Buggy</i> and <i>NotBuggy</i> methods in the <i>Titan</i> project across the <i>Noisy</i> and <i>Less-Noisy</i> datasets. Metric distributions differ more significantly in the <i>Less-Noisy</i> dataset.	46
3.1	Method-level bug prediction pipeline. The pipeline comprises four phases: (1) data collection, which constructs four datasets (Inception/Evolution $\times$ Noisy/Less-Noisy); (2) feature extraction using software metrics and code embeddings; (3) classification with a two-tower fusion network and traditional baselines; and (4) evaluation under project and temporal splits using imbalance-aware metrics.	60
3.2	Number of <i>NotBuggy</i> and <i>Buggy</i> methods, and the percentage of buggy methods. We further resample all the dataset so that both variant in a family has same number of samples.	64
3.3	Test-set metrics of the baselines and best fusion model on the <i>Inception</i> dataset. Results improve gradually, and fusion leads on most metrics.	77

3.4	Five-fold cross-validation of the best fusion model on the full <i>Inception</i> Less-Noisy dataset (≈ 20 projects per fold). The wide fold-to-fold spread shows that project composition, not model quality, drives much of the variation in reported performance.	78
3.5	CDF of per-project metrics for the best fusion model on the 20 test projects. All projects outperform trivial baselines ($MCC > 0$, $ROC-AUC > 0.5$, $PR-AUC$ above the <i>Buggy</i> rate), but performance varies widely. For readability the number of markers is fewer than the number of data points.	79
3.6	Test-set metrics for baselines and the best fusion model on the Evolution family. Fusion and the tabular less-noisy baseline have similar performance, with fusion providing higher buggy recall.	85
3.7	Five-fold cross-validation of the best fusion model over the full Evolution Less-Noisy dataset (≈ 20 projects per fold). The wide fold-to-fold spread shows that project composition, not model quality, drives much of the variation in reported performance.	86
3.8	CDF of per-project metrics for the best fusion model on the 20 Evolution test projects. Nineteen of twenty clear $MCC > 0$, but the median per-project MCC is only 0.05. For readability the number of markers is fewer than the number of data points.	87
3.9	Per-project MCC and buggy rate for the fusion model on the <i>Inception</i> dataset. Buggy rate, not sample size, drives MCC	89
4.1	Overview of methodology that involves app collection (with source code and ratings), feature extraction, feature selection, and building predictive models.	109
4.2	Cumulative distribution functions for (a) the size (LOC) of the applications in log scale and (b) the number of Java files per application in log scale.	113
4.3	Distribution of apps by <i>genre</i> , with counts shown as bars and the cumulative percentage indicated by the line. For clarity, the 13 game-related genres are aggregated into a single <i>GAME</i> category.	127
4.4	Cumulative Distribution Functions (CDFs) of actual and predicted ratings. Approximately 70% of the actual ratings fall between 3.0 and 4.5, while over 95% of the predicted ratings lie within this range, indicating the model's bias towards predicting around the mean value.	139
4.5	Cumulative Distribution Functions (CDFs) of absolute error and squared error for the Ridge regression model in predicting actual ratings. The plots indicate that a small number of data points account for the majority of the large errors.	140
4.6	Scatter plot of true ratings and predicted ratings for each app.	141
4.7	Scatter Plot of True Ratings vs Predicted Ratings.	141

4.8	Confusion matrix and AUC-ROC curve for rating-based popularity prediction using MLP classifier with SMOTE.	147
4.9	CDF of <i>DownloadsPerYear</i>	148
4.10	CDF of <i>DownloadsPerYear</i> after removing outliers.	148
4.11	CDF of absolute error for the Ridge regression model in predicting <i>DownloadsPerYear</i>	150
4.12	Scatter plot of true <i>DownloadsPerYear</i> vs predicted <i>DownloadsPerYear</i>	151
4.13	Confusion matrix and AUC-ROC curve for <i>DownloadsPerYear</i> -based popularity prediction using MLP classifier.	153

List of Tables

2.1	Performance comparison of LLMs for Diff-only and Diff+Message. Metrics are calculated by targeting the buggy samples (pos_label = “Buggy”). For each scenario, the best value for each metric is highlighted in bold.	33
2.2	Performance comparison of different prompting techniques. Metrics are calculated by targeting the buggy samples (pos_label = “Buggy”).	35
2.3	Predictions on a semantically misleading commit using chain-of-thought + few-shot prompting with <i>GPT-4o</i> .	37
2.4	Performance comparison of ML models trained on <i>text-embedding-004</i> embeddings. Metrics are calculated by targeting the buggy samples (pos_label = “Buggy”).	41
2.5	Performance comparison of MLP classifier trained on different LLM embeddings. The first row of <i>text-embedding-004</i> is taken from Table 2.4 for comparison. Metrics are calculated by targeting the buggy samples (pos_label = “Buggy”).	42
2.6	Cliff’s Delta effect size values compare <i>Noisy</i> and <i>Less-Noisy</i> datasets, grouped by effect size category and shown as percentages in the format <i>Noisy / Less-Noisy</i> . The percentages for the <i>Noisy</i> dataset do not sum to 100% because the <i>RxJava</i> project was excluded as it had no <i>NotBuggy</i> methods.	48
3.1	Representative sample of studied subject systems spanning the range of popularity and size.	61
3.2	Feature sources and their availability.	65
3.3	Top performing configurations on the validation set for the <i>Inception Noisy</i> dataset.	73
3.4	Top performing configurations on the validation set for the <i>Inception Less-Noisy</i> dataset.	75
3.5	Top performing configurations on the validation set of <i>Inception Less-Noisy</i> dataset using Fusion models.	76
3.6	Top performing configurations on the validation set for the <i>Evolution Noisy</i> dataset.	81

3.7	Top performing configurations on the validation set for the <i>Evolution Less-Noisy</i> dataset.	82
3.8	Top performing configurations on the validation set of <i>Evolution Less-Noisy</i> dataset using Fusion models.	84
3.9	Scores on test set for different type of temporal split on the Evolution Less-Noisy dataset. 56.41% of buggy samples from the entire dataset is in early 30% of the data.	86
4.1	Seventeen System-Level Metrics Extracted Using the DV8 Tool. . . .	115
4.2	Class-Level Metrics and Descriptions.	117
4.3	A Subset of Method-Level Metrics and Descriptions.	121
4.4	A Subset of Code Smells and Descriptions.	124
4.5	Metadata Description of Selected App Features.	128
4.6	Regression metrics (RMSE, MAE, R^2) of rating regressors evaluated using three feature selection strategies: Size-only, Handpicked, and Voting. Bold values indicate the best metric scores within each feature set.	137
4.7	Performance metrics (Macro F1, MCC, AUC) of rating-based popularity classifiers evaluated using three feature selection strategies: Size-only, Handpicked, and Voting. Bold values indicate the best metric scores within each feature set.	144
4.8	Classification performance for rating-based popularity prediction using MLP classifier with SMOTE.	146
4.9	Regression metrics (RMSE, MAE, R^2) of <i>DownloadsPerYear</i> regressors evaluated using three feature selection strategies: Size-only, Handpicked, and Voting. Bold values indicate the best metric scores within each feature set.	149
4.10	Classification metrics (Macro F1 Score, MCC, AUC) of <i>DownloadsPerYear</i> classifiers evaluated using three feature selection strategies: Size-only, Handpicked, and Voting. Bold values indicate the best metric scores within each feature set.	152
4.11	Classification performance for <i>DownloadsPerYear</i> -based popularity prediction using MLP classifier.	153

Chapter 1

Introduction

Software systems must continuously evolve to remain useful. They are modified to accommodate new requirements, repair defects, improve performance, adapt to changing platforms, and respond to evolving user expectations [133]. As a result, software maintenance is often the longest and most expensive phase of the software development life cycle, sometimes exceeding the cost of initial development [69, 116]. Within this maintenance process, activities such as feature implementation, bug fixing, refactoring, and quality assurance require developers to make repeated decisions about where limited effort should be spent.

Prediction models are one way to support such decisions. If developers can identify risky code components, likely defects, or weak release candidates early, they can prioritize inspection, testing, refactoring, and other quality-improvement activities more effectively [158, 257]. By learning from existing software artifacts and historical development data, such models can estimate future software outcomes before they become costly to observe directly. For example, if a model can identify bug-

prone code components early, developers may allocate testing and review effort more effectively. Similarly, if a model can estimate quality-related outcomes before deployment, developers may improve implementation and release decisions. External software metrics—such as correctness, and performance—can predict future maintenance effort, but they are difficult to collect and are often not available in early development phases [83]. In contrast, software metrics are easy to collect and are available throughout the software development life cycle. Therefore, a holy grail for the developer and the research community has been to predict future maintenance effort from software metrics [50, 83].

Software-metric-based prediction is therefore a long-standing research direction in software engineering [73, 190, 47, 50]. It is motivated by a practical goal: using measurable internal properties of software to anticipate future quality, maintenance, or user-facing outcomes. However, the success of this goal is not guaranteed. The predictive value of software metrics depends on what is being predicted, at what granularity, from what data, and under what evaluation protocol [50]. This thesis examines this broader question by studying the limits and promise of software-metric-based prediction across fine-grained software quality prediction and launch-time app popularity prediction.

1.1 Software-Metric-Based Prediction

Software metrics are quantitative measurements that describe properties of software artifacts, development processes, or developer activity [190, 209]. They can be collected at different granularities, including system, file, class, method, commit,

and line levels [190, 81, 119, 181, 14]. Common examples include source lines of code, cyclomatic complexity, coupling, cohesion, fan-in, fan-out, change frequency, code churn, developer ownership, and bug-history measures [153, 46, 21, 190]. These metrics are commonly used because they are relatively inexpensive to collect and can summarize aspects of software structure and evolution that may be related to future quality outcomes [158, 190, 50].

Metric-based prediction has been widely studied in software engineering [88, 136, 195, 165, 158, 47, 51]. In defect prediction, for example, metrics are used to estimate whether a software component is likely to contain bugs [257, 11, 21, 24, 83, 219, 241, 158, 190]. In maintenance research, code metrics have been studied as indicators of change-proneness and maintenance effort [50]. In mobile software engineering, internal code-quality indicators have also been studied in relation to external outcomes such as app ratings, market success, and user-perceived quality [58, 38, 80, 23, 222].

Despite this long history, metric-based prediction remains difficult. First, prediction quality depends heavily on dataset quality. Second, evaluation design strongly affects reported performance. Third, software metrics may behave differently depending on prediction granularity, project context, and target outcome [50]. Therefore, software-metric-based prediction must be evaluated with attention to label quality, temporal realism, project generalization, class imbalance, and the practical meaning of the prediction task.

1.2 Motivation

Software-metric-based prediction is attractive because it promises early, automated, and relatively inexpensive support for software engineering decisions [190, 50]. If internal metrics can reliably signal future defects or maintenance needs, developers can prioritize inspection, testing, refactoring, and other quality-assurance activities before problems become more expensive to address [158, 257]. However, prior research shows that the usefulness of software metrics is not universal. Their predictive value varies with the target outcome, evaluation context, software granularity, and characteristics of the underlying data [83, 50, 190]. In particular, prediction quality depends on the quality of the underlying data, the granularity of prediction, the evaluation protocol, and the nature of the target outcome [50, 51].

This thesis is motivated by this tension. Rather than treating software metrics as either generally powerful or generally ineffective, it studies their predictive value under increasingly demanding conditions. The first study asks whether method-level bug datasets can be improved by detecting tangled code changes. The second study asks how far method-level bug prediction can go once label quality, feature richness, and evaluation realism are considered together. The third study widens the lens and asks whether internal metrics can predict app popularity at launch, where the target is not an internal quality property but an external, user-facing outcome. A shared thread runs through all three: prediction at inception. Whether the unit is a method at its introduction or an app at its first release, we ask what internal metrics can reveal before the outcome becomes costly to observe. Together, these studies provide a context-sensitive investigation of the limits and promise of software-metric-based

prediction.

1.2.1 Method-Level Bug Prediction

Method-level bug prediction is practically attractive because it offers a middle ground between coarse-grained and overly fine-grained prediction. File- and class-level predictions may still leave developers with large regions of code to inspect [180, 51], whereas line-level predictions can be noisy because individual lines often require surrounding context [181, 241]. Methods are localized enough to support developer action while still preserving meaningful semantic context [81, 51].

However, a central obstacle in method-level bug prediction is label noise. Bug datasets are often constructed from bug-fix commits, where methods modified in a bug-fix commit are labeled as buggy. This assumption becomes unreliable when commits are tangled, because a single commit may combine an actual bug fix with unrelated refactorings, cleanups, or feature changes [103, 104]. As a result, methods that did not contribute to the bug fix may be incorrectly labeled as buggy. This motivates the first study, which investigates whether large language models can identify whether a method-level code change is actually related to the bug-fixing intent of a commit.

Prior studies also show that earlier method-level bug prediction results often weaken under realistic evaluation, making the problem still open [180, 182, 51]. Furthermore, previous studies only relied on code metrics and did not consider learned code representation (embeddings). These motivates the second study, which evaluates how far method-level bug prediction can go when cleaner labels, traditional

software metrics, learned code representations, and realistic evaluation protocols are considered together.

1.2.2 App Popularity Prediction

While bug prediction concerns internal software quality, developers must also reason about external outcomes such as user reception and market success. For mobile applications, user ratings and download counts are widely used indicators of popularity, and predicting these outcomes before an app’s first release could provide developers with an opportunity to improve implementation and release decisions before user feedback becomes available [96, 95]. This launch-time setting is particularly important because early popularity signals can affect an app’s ranking and visibility in app stores [120].

However, app popularity is a substantially broader outcome than traditional software quality. It may depend not only on the internal characteristics of the source code but also on factors such as app features, user interface quality, genre, advertising, permissions, and marketing [222, 90, 230, 198]. Accordingly, prior studies report inconsistent relationships between internal software quality and app popularity, ranging from weak or no associations to evidence that code-related characteristics matter in particular contexts [58, 80, 59]. Moreover, the closest prior prediction work relied on historical app data and evaluation settings that do not directly support prediction at an app’s inception [38].

These gaps motivate the third study, which asks to what extent internal software metrics available at launch can predict an app’s future popularity. Rather than

assuming that internal metrics should precisely determine user-facing outcomes, the study evaluates how much predictive signal they provide for ratings and downloads. In doing so, it extends the software-metric-based prediction debate beyond traditional maintenance outcomes and examines the limits and promise of internal metrics in a broader, user-facing context.

1.3 Contributions

The contributions of this thesis are based on the promises of software-metric-based prediction and their limitations. This section summarizes the contributions in two research thrusts: testing the limits of method-level bug prediction, and assessing the feasibility of app popularity prediction.

1.3.1 A Realistic Evaluation of Method-Level Bug Prediction

We approach method-level bug prediction by confronting its well-known problems: noisy data, shallow features, and unrealistic evaluation. First, we propose a Large Language Model (LLM) based technique that removes tangled noise from method-level bug datasets. This creates the foundation for our second study, which feeds the resulting higher-quality dataset, together with software metrics and code embeddings, into a realistic evaluation pipeline and shows that method-level bug prediction is still an open research problem. The first study is thus the enabler and the second is the test: Chapter 2 builds the cleaner data that Chapter 3 pushes to its limits. This contribution is an amalgamation of two sub-contributions:

LLM-Based Detection of Tangled Code Changes for Higher Quality Method-Level Bug Datasets

First, this thesis contributes an LLM-based approach for detecting tangled code changes at the method level. The approach uses commit messages and method-level code diffs to determine whether a changed method is actually related to the bug-fixing intent of a commit. This contribution addresses tangled changes, a key source of label noise, in method-level bug datasets. By reducing such false-positive labels, the study provides a stronger empirical foundation for method-level bug prediction. Our best model detects tangled changes with an F1-score of 0.906, and the resulting *Less-Noisy* dataset makes commonly used code metrics substantially better at separating buggy from non-buggy methods across 49 open-source projects.

This study was published as a technical paper in The 23rd International Conference on Mining Software Repositories (MSR 2026).

Method-Level Bug Prediction: How Far Are We?

Second, this thesis contributes a large-scale empirical evaluation of method-level bug prediction under realistic conditions. The study evaluates prediction in two settings: at method inception and across method evolution. It combines traditional code metrics with developer, change-history, bug-history, and code-embedding features, and evaluates models using project-wise and temporal splits. The results show that cleaner labels and richer features help consistently, lifting inception-time MCC from 0.13 to 0.18 at best, but they are not sufficient to make method-level bug prediction practically reliable. The study identifies severe class imbalance and weak

cross-project generalization as the major remaining barriers.

This study has been submitted as a research paper and currently under review in a reputed international conference on software engineering.

1.3.2 A Launch-Time Evaluation of App Popularity Prediction Using Internal Metrics

Third, this thesis contributes an empirical study of app popularity prediction at launch using internal software metrics. The study constructs a dataset of open-source Android applications by linking source-code information from F-Droid with popularity indicators from Google Play. It evaluates whether system-, class-, and method-level metrics, code smells, and app metadata can predict app ratings and downloads. The findings show that exact popularity prediction remains difficult, but internal metrics provide moderate signal when the task is framed as classification, reaching an F1-score of 0.72 for ratings. This contribution extends the software-metric-based prediction debate beyond traditional maintenance outcomes and shows that metric usefulness depends strongly on prediction context.

This study has been submitted as a research paper and currently under review in a reputed journal on software engineering.

1.4 Thesis Organization

The remainder of this thesis is organized as follows.

Part I focuses on method-level bug prediction. Chapter 2 presents the first study,

which investigates LLM-based detection of tangled code changes for constructing higher-quality method-level bug datasets. Chapter 3 presents the second study, which evaluates the limits of method-level bug prediction under cleaner labels, richer features, and realistic evaluation protocols.

Part II focuses on app popularity prediction. Chapter 4 presents the third study, which investigates whether internal software metrics available at launch can predict app ratings and downloads.

Finally, Chapter 5 concludes the thesis by summarizing the overall findings, discussing the implications for software-metric-based prediction, and outlining future research directions for method-level bug prediction and app popularity prediction.

Part I

Method-Level Bug Prediction

Part I asks how far internal software metrics can take method-level bug prediction once its data problem is addressed. Chapter 2 builds the foundation: an LLM-based technique that detects tangled code changes and removes them from method-level bug datasets. Chapter 3 then puts that foundation to the test, combining the cleaner labels with richer features under realistic evaluation to establish how far the prediction itself can go.

Chapter 2

LLM-Based Detection of Tangled Code Changes for Higher-Quality Method-Level Bug Datasets

Abstract

Tangled code changes, commits that conflate unrelated modifications such as bug fixes, refactorings, and enhancements, introduce significant noise into bug datasets and adversely affect the performance of bug prediction models. Addressing this issue at a fine-grained, method-level granularity remains unexplored. This is critical to address, as recent bug prediction models, driven by practitioner demand, are increasingly focusing on finer granularity rather than traditional class- or file-level predictions. This study investigates the utility of Large Language Models (LLMs) for detecting tangled code changes by leveraging both commit messages and method-level code

diffs. We formulate the problem as a binary classification task and evaluate multiple prompting strategies, including zero-shot, few-shot, and chain-of-thought prompting, using state-of-the-art proprietary LLMs such as *GPT-5* and *Gemini-2.0-Flash*, and open-source models such as *GPT-OSS-120B* and *CodeBERT*.

Our results demonstrate that combining commit messages with code diffs significantly enhances model performance, with the combined few-shot and chain-of-thought prompting achieving an F1-score of 0.883. Additionally, we explore machine learning models trained on LLM-generated embeddings, where a Multi-Layer Perceptron classifier achieves superior performance (F1-score: 0.906, MCC: 0.807). Applying our approach to 49 open-source projects improves the distributional separability of code metrics between buggy and non-buggy methods, demonstrating the promise of LLMs for method-level commit untangling and potentially contributing to improving the accuracy of future bug prediction models.

2.1 Introduction

Software maintenance often exceeds the cost of initial development [69, 116], with bug detection and resolution alone consuming 50–70% of total development expenses [233]. Undetected bugs can result in cascading failures, reduced developer productivity, and increased technical debt [54, 214]. Consequently, the research community has extensively investigated bug prediction techniques to facilitate timely and targeted maintenance interventions [51, 182, 234, 81, 193]. Early bug prediction models focused on class- or file-level granularity [257, 11, 21, 24, 83, 219, 241]. However, these models are less helpful in practice, as only a small fraction of code in large

units is typically defective [161, 182]. Line-level models offer finer granularity but are prone to false positives due to coincidental line similarity [51, 202, 89]. This has driven growing interest in method-level bug prediction [51, 161, 182, 210, 81, 98, 158], which aligns more closely with real-world debugging practices [89, 51]. However, method-level bug prediction faces challenges including evaluation bias and noisy data [182, 51], and progress is constrained by the scarcity of clean, labeled datasets [51, 182].

A key obstacle in building clean bug datasets is the presence of *noise* in training data [28], often caused by *tangled code changes* [51, 179, 104]. While Version Control Systems like Git are intended for atomic commits, developers frequently combine unrelated tasks, such as bug fixes, refactorings, and enhancements, into a single commit [103, 65, 126], violating best practices and introducing tangled changes. Researchers typically use commit messages to identify bug-fix commits, assuming all associated modifications are relevant. However, this assumption fails in the presence of tangled commits, leading to false positives in labels, which degrades dataset quality and impairs bug prediction performance [104, 101].

To mitigate the impact of tangled changes, several prior works have proposed various untangling techniques ranging from heuristic-based program slicing [164], lexical and data-flow based models [179], and graph-based learning using dependency and name flows [138, 237]. While effective at the file or statement level, none of these methods have addressed the problem at the method-level granularity, leaving a gap that presents a unique opportunity for exploration. Method-level untangling cannot be achieved using file-level approaches, as a single file may contain multiple methods, only some of which are bug-related. Labeling all methods in a file as buggy

is inappropriate and would produce numerous false positives. On the other hand, although statements can be aggregated to form method-level representations, determining whether an individual statement contributes to a bug fix requires understanding its broader context, including surrounding code, method semantics, and commit intent [224, 213]. For developers and models alike, the relevant context may span from within the method to the entire file or even across files [213]. For instance, in the *refill* function of *FixedIntervalRateLimiter.java* from *Apache HBase Commit 84a5039*¹, the function signature modification appears to be a bug fix when viewed in isolation, as removing an argument typically alters the function’s logic; however, examining the full method reveals it is actually a refactoring change. Thus, simply applying a statement-level model repeatedly at the method-level would require redundant context analysis for each modified line, introducing unnecessary computational overhead. Therefore, distinct representations at different granularity are needed to effectively model bug-fix behavior while maintaining contextual relevance.

Given that, this study focuses on detecting tangled changes at the method-level by framing the task as a binary classification problem: determining whether a method-level code change is related to a bug fix (*Buggy*) or not (*NotBuggy*) based on commit messages and code diffs. To tackle this, we explore the use of *Large Language Models (LLMs)*, which have demonstrated strong capabilities in analyzing natural language and source code [44, 167]. In contrast to prior approaches in code analysis, classification, and untangling that rely on language-specific algorithms [104, 91] or complex models requiring extensive training and feature engineering [243], LLM-based methods enable faster system development through their language-agnostic design. LLMs

¹<https://github.com/apache/hbase/commit/84a50393ee56d09abb68f54b44b64f5279bd33c9>

also provide advantages such as promptability, flexibility in input handling, and the ability to leverage contextual signals from both code and commit metadata.

We evaluate several prompting strategies using both proprietary and open-source LLMs. Additionally, we develop machine learning models using LLM generated embeddings. Finally, we evaluate the impact of the LLM-based technique by measuring improvements in the distributional separability of code metrics between *Buggy* and *NotBuggy* methods through statistical analysis. In general, the contribution of this study is centered around four research questions.

RQ1: *Can zero-shot LLMs detect tangled code changes by using code diff and commit message?*

We designed a classification-based approach that uses LLMs to detect tangled changes from method-level code diffs, using zero-shot prompting, with or without the corresponding commit messages. Through experiments on our curated gold dataset, we show that including the commit message significantly improves the performance across multiple LLM variants, achieving the highest F1-score of 0.879 using *GPT-5*.

RQ2: *How do different prompting techniques influence the effectiveness of detecting tangled code changes?*

We investigate the impact of advanced prompting strategies, such as few-shot, chain-of-thought, and a hybrid combination. Our analysis shows that LLMs can reason more accurately about change semantics when guided with structured prompts and examples. Among these, chain-of-thought + few-shot prompting with *GPT-4o* achieves the most balanced performance (F1-score: 0.883) and demonstrates superior capability in handling complex inputs.

RQ3: *How well can embedding-based machine learning models detect tangled code changes?*

We utilized LLMs to generate embeddings from commit messages and code diffs, which served as input for embedding-based classifiers. One of the models exhibits strong predictive capabilities, outperforming the results obtained in RQ1 and RQ2. Specifically, a Multi-Layer Perceptron classifier achieved the highest F1-score of 0.906, demonstrating the effectiveness of embedding-based representations for detecting tangled changes.

RQ4: *What is the potential impact of LLM-based untangling on future method-level bug prediction models?*

Using a *Less-Noisy* dataset, created by filtering out noisy samples through our LLM-based approach, we observe that the distributional differences in various code metrics between *Buggy* and *NotBuggy* methods are significantly more than in the original noisy dataset. This finding shows promise for future machine learning (ML) models for bug prediction, as many of them rely on these code metrics to predict whether a method is bug-prone or not.

To enable replication, we share our data and code publicly.²

2.2 Related Work

This section reviews prior work on tangled code changes, their prevalence, impact, and untangling techniques, as well as LLM-based approaches for analyzing and classifying text and code. Together, these studies motivate our research.

²<https://github.com/SQMLab/Tangled>

2.2.1 Tangled Changes

Herzig and Zeller [103] first used the term *Tangled Changes* and reported that up to 15% of bug-fixing commits across several open-source Java projects contained tangled changes. Subsequently, Tao and Kim [221] later observed tangling in up to 29% of revisions. Herbold et al. [101] confirmed its prevalence and impact, showing that only 22–38% of lines in bug-fix commits actually fix bugs. Kochhar et al. [127] further found that 28% of files in bug-fix commits contain no bugs.

The prevalence of tangled changes greatly affects the historical analysis of source code. By labeling all the modified methods in a bug-fixing commit as buggy methods, it harms the bug prediction models significantly [104, 51]. Consequently, researchers focused on detecting and untangling commits [103, 221, 91, 164, 179].

Early techniques relied on static analysis and heuristics. Herzig et al. [103] introduced confidence voters based on file distance, change coupling, call graphs, and data dependencies to estimate task co-membership. Tao and Kim [221] applied program slicing and pattern matching to cluster semantically related edits, achieving 69% agreement with manual decompositions. Guo and Song [91] proposed CHGCUTTER, an interactive method using control and data dependencies to decompose composite changes while preserving syntactic correctness. Muylaert and De Roover [164] applied program slicing on abstract syntax trees to untangle composite commits by grouping fine-grained code changes according to their dependence graph slices. Pârtachi et al. [179] introduced Flexeme, which overlays name flows on program dependency graphs using a δ -NFG structure to better capture lexical cues. Using Agglomerative clustering for commit untangling they developed a tool Heddle, which achieved

improved clustering accuracy (F1-score of 0.81) and runtime efficiency.

Graph-based modeling has emerged as a powerful approach for capturing complex code relationships. Shen et al. [207] proposed SmartCommit, representing diff hunks as graph nodes enriched with semantic links (hard, soft, refactoring, cosmetic), and achieved a decomposition accuracy between 71-84%. Chen et al. [41] and Xu et al. [237] enhanced this approach by incorporating node attributes such as token content and control/data flow, achieving 7.8% and 8.2% higher accuracy than Flexeme, respectively.

Fan et al. [70] introduced a Heterogeneous Directed Graph Neural Network designed to capture semantic dependencies without relying on explicit code links. By leveraging hierarchical graphs at the entity and statement levels, they achieved substantial untangling improvements, 25% for C# and 19.2% for Java, outperforming Flexeme and SmartCommit without sacrificing time efficiency.

Recent work increasingly applies machine learning to learn untangling patterns from data. UTANGO [138] uses a Graph Convolutional Network to learn contextual embeddings of code changes, incorporating cloned code and surrounding context. Framed as supervised clustering, it outperforms Flexeme by 9.9% accuracy. Dias et al. [65] and Liu et al. [144] explored fine-grained tracking and clustering based on developer interactions and commit metadata to achieve high temporal and contextual resolution.

Although these methods have shown success at the file- or statement-level, the method-level remains unexplored, despite being preferred by practitioners and researchers in order to create a clean and noise-free method-level bug dataset [182, 51].

Motivated by this gap, we investigate whether tangled changes can be detected from method-level diffs and their corresponding commit messages.

2.2.2 LLMs for Code Analysis & Classification

LLMs have shown strong performance in natural language understanding and source code analysis [31, 253]. Trained on large corpora of text and code, they generalize well to unseen inputs and often perform effectively in zero-shot settings [128]. This versatility has fueled growing interest in using LLMs for text classification and code analysis [5, 72]. Building on these capabilities, our study explores LLMs for detecting tangled code changes by jointly analyzing commit messages and code diffs.

Prompt design is critical to optimizing LLM performance. Recent advances in prompt engineering have introduced zero-shot, few-shot, and chain-of-thought prompting [235, 31, 236]. Zero-shot learning uses only task instructions, while few-shot prompting includes exemplar input-output pairs to guide responses, yielding improved performance across tasks [31]. Chain-of-thought prompting enhances reasoning by encouraging step-wise problem decomposition [236]. Motivated by their empirical success, we systematically evaluate these techniques for detecting tangled code changes.

LLMs are built on Transformer architectures, consisting of encoder and decoder modules [225]. Encoders convert input text into high-dimensional embeddings that capture semantic relationships, making them effective for downstream classification [186, 121]. We extend this utility to software engineering by generating embeddings from commit messages and code diffs, and training machine learning models to classify them as tangled or untangled.

LLM embeddings are also used in commit classification tasks [244, 243], which classifies commits into maintenance activity types. CodeBERT [72] has been used in COLARE [243] to classify commits into corrective, adaptive, and perfective categories by integrating hunk-level code representations with commit messages and file features. However, such models are built for commit-level classification which does not properly align with our method-level objectives.

In summary, this study is motivated by three factors: (1) the lack of fine-grained research on the tangled changes at the method level, especially for bug prediction [104, 51]; (2) the proven capabilities of LLMs in analyzing textual and code artifacts via advanced prompting [31, 253, 235, 236]; and (3) the semantic richness of LLM-generated embeddings for classification [186, 121]. *To the best of our knowledge, this is the first study to evaluate LLMs for detecting tangled code changes within bug-fix commits at the method level.*

2.3 Methodology

Figure 2.1 shows the overview of our approach that we use to evaluate LLMs in distinguishing *Buggy* diff and *NotBuggy* diff. In this section, the components of the methodology are described step by step.

2.3.1 Dataset

To assess whether LLMs can be used to untangle real bug fix changes from other changes, a dataset is required that includes both genuine bug fix code diffs and non-bug-fix code diffs. We provide the model with a code diff with or without the commit

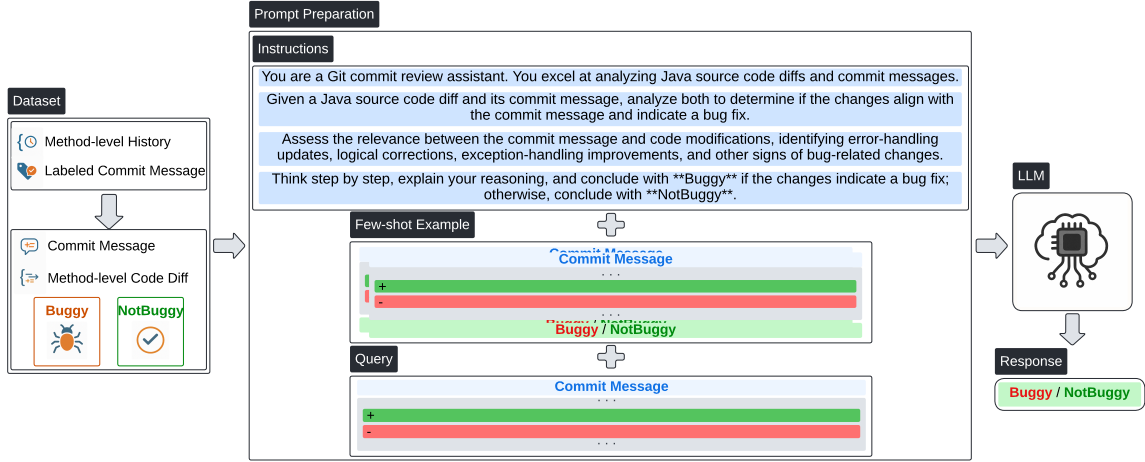


Figure 2.1: The methodology for $RQ1$ and $RQ2$ consists of three parts: (a) dataset, (b) prompt preparation, and (c) LLM Query. The prompt setup for few-shot + chain-of-thought prompting consists of four instruction parts: (i) persona, (ii) task description, (iii) behavioral guidance, and (iv) output formatting. For Diff-only detection of $RQ1$, references to the commit message are excluded from everywhere. Other experiments use the full instructions, with the output formatting modified to require a single-word answer in non-chain-of-thought setups. Few-shot examples are included only in few-shot experiments.

message to evaluate its ability to distinguish between bug-fix and non-bug-fix changes.

Fortunately, an existing dataset by Chowdhury et al. [51] contains 774,051 Java methods from 49 open-source projects, where each method is associated with its complete change history and is enriched with metadata such as commit messages, timestamps, authorship, and change type. The dataset also includes code diffs for all historical changes, labels indicating whether each change is related to a bug fix or not, and information on the number of methods modified within the same commit.

However, the dataset contains noise. In some cases, methods involved in a bug fix commit may not contribute to the actual fix, representing instances of tangled changes. We use this dataset to construct a curated gold dataset by removing such noise. The construction process involves a two-step sampling strategy, combining automated filtering and manual labeling, to ensure high quality and accurate representation of tangled change scenarios.

Automated Approach. The dataset provided by Chowdhury et al. [51] includes the total number of methods modified in each commit. They recommended that if a method is the only one modified in a bug fix commit, it can be confidently considered a bug-related change. We classify such code diffs as *Buggy* without ambiguity. Using this criterion, we collected 730 Buggy diffs. They also suggested that a method can be considered *NotBuggy* if it has never appeared in a bug fix commit throughout its lifetime. Based on this guideline and leveraging the large pool of eligible *NotBuggy* methods, we randomly selected 730 samples that have no history of being part of any bug fix commit. During this process, three duplicate diffs, likely caused by duplicated code across codebases, were identified and removed, resulting in 727 unique NotBuggy instances. Thus, the *Automated* step of our gold dataset construction produced a total of 1,457 method-level examples. We intentionally limited the number of NotBuggy samples, as querying proprietary LLMs incurs significant computational and financial costs. Including all such samples would have significantly increased the overall cost.

Manual Approach. While the *Automated* approach ensured high confidence in labeling, it did not capture the complexity of real-world tangled change scenarios, where multiple unrelated modifications often occur within a single commit. To

address this limitation, we introduced a second step focused on incorporating tangled examples. In this step, we extracted additional method-level diffs from bug-fix commits that modified multiple methods. The first author, a graduate student with over two years of industry experience in software engineering, manually labeled each method-level change to determine whether it was related to a bug fix (*Buggy*) or was incidental, such as refactorings or unrelated enhancements (*NotBuggy*). Each method was reviewed in conjunction with its associated commit message to evaluate semantic alignment with bug-fixing intent. The evaluation emphasized structural corrections, error handling logic, and domain-specific repairs. To promote annotation consistency and minimize errors, we developed a user interface that allowed side-by-side inspection of code diffs and their corresponding commit metadata. Using this *manual* approach, we identified 166 tangled *Buggy* methods and 141 tangled *NotBuggy* methods from 450 randomly selected commits in the dataset of Chowdhury et al. [51].

Combining all steps, the gold dataset consists of 1,764 unique method-level change instances, each labeled as either *Buggy* or *NotBuggy*, based on their actual relevance to bug fixing activities. To evaluate labeling reliability, an independent validation was conducted by another graduate student in software engineering, who randomly picked 100 samples from the manually labeled portion of the dataset and labeled them independently. Some disagreements arose, mainly due to differing interpretations of what constitutes a bug. Specifically, the first labeler marked some quality-related fixes as *NotBuggy*, while the second considered them *Buggy*. For example, in a commit ³ in the SonarQube project, the method `JwtHttpHandler#getTokenFromCookie` was modified by replacing `isNullOrEmpty(token)` with `isEmpty(token)`. One annotator initially in-

³<https://github.com/SonarSource/sonarqube/commit/fd78d281d57d7b1ffa49d27cd03de39b6a961592>

terpreted this as a behavioral change; upon inspection of the two helper methods, we found they were semantically equivalent (both reject null and empty strings). The commit message (“Fix Quality flaws”) was also consistent with a quality-only change. Following a discussion with the last author, quality-fix-related changes were finalized as *NotBuggy*. The inter-rater agreement, measured using Cohen’s Kappa [55], was 0.82, indicating strong consistency and confirming the reliability of the labeling process. Consequently, this dataset provides a robust benchmark for both training and evaluating our detection approaches, while also offering a valuable foundation for future research in this domain.

2.3.2 Prompt Preparation

The design of our prompt structure draws on recent advancements in prompt engineering that demonstrate the effectiveness of decomposing complex instructions into clearly scoped components [31, 236, 254]. As prior work has shown, LLMs are highly sensitive to prompt formulation, and performance can vary substantially based on how the task, instructions, and examples are framed [143]. To maximize reliability and adaptability, we designed a modular prompt template consisting of three core components, aligned with prior studies [13, 84], as illustrated in Figure 2.1.

Instructions. This component defines the LLM’s role and behavior. It is composed of four parts: (i) a role-based persona (e.g., “You are a Git commit review assistant”), which helps constrain the model’s responses by grounding its behavior and improves performance [106]; (ii) a high-level task description [13]; (iii) step-by-step behavioral guidance to structure the model’s reasoning [236]; and (iv) explicit

Fix bug in InetAddresses and performance problem in InternetDomainName (changed from regex to CharMatcher). This is intended as the last code update before release 06.

```
@@ -1,18 +1,18 @@
private static boolean validateSyntax(List<String> parts)
{
    final int lastIndex = parts.size() - 1;
    // Validate the last part specially, as it has
    // different syntax rules.
-   if (!validatePart(parts.get(lastIndex), FINAL_PART)) {
+   if (!validatePart(parts.get(lastIndex), true)) {
        return false;
    }
    for (int i = 0; i < lastIndex; i++) {
        String part = parts.get(i);
-       if (!validatePart(part, NORMAL_PART)) {
+       if (!validatePart(part, false)) {
            return false;
        }
    }
    return true;
}

@@ -1,12 +1,12 @@
private static String convertDottedQuadToHex(String ipString) {
    int lastColon = ipString.lastIndexOf(':');
    String initialPart = ipString.substring(0, lastColon + 1);
    String dottedQuad = ipString.substring(lastColon + 1);

    byte[] quad = textToNumericFormatV4(dottedQuad);
    if (quad == null) {
        return null;
    }

-   String penultimate = Integer.toHexString((quad[0] << 8) | (quad[1] & 0xff));
-   String ultimate = Integer.toHexString((quad[2] << 8) | (quad[3] & 0xff));
+   String penultimate = Integer.toHexString(((quad[0] & 0xff) << 8) | (quad[1] & 0xff));
+   String ultimate = Integer.toHexString(((quad[2] & 0xff) << 8) | (quad[3] & 0xff));

    return initialPart + penultimate + "." + ultimate;
}
```

Figure 2.2: Commit message and two methods modified in the same commit: the message indicates a bug fix, but only the method on the right contains the bug-fix change, while the method on the left is a non-bug-related tangled change. This was used as few-shot examples in our prompts.

output formatting requirements to reduce ambiguity and improve parsability [145].

Few-shot Examples. Few-shot prompting has been shown to significantly improve performance by enabling in-context learning, especially in code and software engineering tasks [253]. For few-shot examples, as shown in Figure 2.2, we selected a representative bug-fix commit containing one diff genuinely related to a bug fix and another unrelated change, a real instance of tangled code changes. For the negative example (*NotBuggy*), we selected the code diff that merely replaced constant values with equivalent boolean literals, without altering the program’s logic or behavior. In contrast, the other code diff (*Buggy*) addressed an issue in the byte-to-hexadecimal conversion logic by correctly handling signed bytes using bit masking, rectifying a functional error. These two fixed examples were consistently utilized to prime the LLMs in all experiments employing few-shot learning. These examples were selected to represent contrasting scenarios, one clearly indicative of a bug-fix and the other

representing a refactor or performance-related change that does not qualify as a bug-fix, although both of the examples come from the same commit.

LLMs often exhibit significant performance degradation as prompt length increases, due to limitations in handling extended context [135, 248]. Additionally, longer prompts incur higher computational and monetary costs, making token efficiency a practical concern. Since method-level diffs can vary greatly in length and the input query, over which we have no control, is the primary driver of prompt size, we deliberately restricted the prompt to a single example per class (*Buggy* and *Not-Buggy*). This prompting approach helps maintain prompt lengths within manageable token limits across different model families. By limiting the number of in-context examples, we strike a balance between performance and resource usage. This strategy aligns with established practices in few-shot prompting [31], where concise and diverse instances are used to anchor the model’s reasoning prior to inference.

Query. This includes the method-level code diff and its corresponding commit message.

Each research question required slight variations in prompt configuration, e.g., enabling or disabling few-shot examples, toggling chain-of-thought reasoning, or isolating input modalities (Diff-only vs. Diff+Message). All corresponding prompt templates are included in our replication package, located within the *prompts* directory.

2.3.3 Selecting and Querying LLMs

Selecting LLMs. For RQ1 and RQ2, we selected six LLMs: four proprietary, *GPT-4o-mini*, *GPT-4o*, *GPT-5*, and *Gemini-2.0-Flash* and two open-source, *GPT-*

OSS-20B and *GPT-OSS-120B*. The proprietary LLMs were chosen based on their performance on recent code-related benchmarks [86, 174, 22, 32], robust API support, and cost-effectiveness for large-scale experimentation. *GPT-4o* serves as OpenAI’s full-scale model offering superior robustness over its smaller variant, *GPT-4o-mini* [174, 32], while *GPT-5* represents the latest generation of OpenAI’s LLMs. *Gemini-2.0-Flash*, from Google [86], delivers competitive performance with the GPT series and has demonstrated effectiveness in code-related tasks [22].

For open-source alternatives and to ensure reproducibility and accessibility, we choose *GPT-OSS-20B* (requiring 16 GB of GPU memory) and *GPT-OSS-120B* (runnable on a single 80 GB GPU) [3]. Despite their moderate scale, these models deliver reasoning and code-understanding performance comparable to proprietary baselines [3, 27], balancing efficiency and capability.

For RQ3, we employed four models for embedding generation: one proprietary and three open-source. The proprietary model, Gemini’s *text-embedding-004* offers high-quality embeddings optimized for classification tasks [132]. Open-source models, *CodeBERT* [72] and *GraphCodeBERT* [92] are widely adopted for code representation learning [146], while *EmbeddingGemma-300m*, a recent lightweight model from Google DeepMind, is designed for low-latency, high-throughput applications [226]. The models used in RQ1 and RQ2 were not included in RQ3 because they are decoder-only generative models optimized for text generation, whereas RQ3 required encoder-based architectures specialized for producing fixed-length embeddings suitable for classification tasks.

In summary, proprietary models were selected for their state-of-the-art perfor-

mance across benchmarks, including code-related tasks [22, 18], while open-source models were selected for their competitive accuracy and efficiency relative to larger open-source alternatives.

Querying LLMs. The three parts of the prompt (instruction + examples + query) are concatenated and formatted according to the input constraints of the respective LLM. The final query prompt is submitted to the LLM, which classifies the change as either *Buggy* or *NotBuggy*. Outputs are reviewed and filtered automatically, if required, to ensure alignment with evaluation criteria.

2.3.4 Evaluation Metrics

We evaluate model performance using standard classification metrics, including accuracy, precision, recall, and F1-score. Additionally, we use the Matthews Correlation Coefficient (MCC), a robust metric that considers all four outcomes: true positives, true negatives, false positives, and false negatives. Unlike accuracy, MCC provides a more balanced measure. An MCC of 1 indicates perfect prediction, 0 indicates random guessing, and -1 reflects total disagreement between predictions and actual labels.

2.3.5 Experimental Setup

All experiments involving open-source LLMs were conducted on a High-Performance Computing (HPC) cluster equipped with Intel 8570 CPUs (2.1 GHz) and multiple NVIDIA H100 GPUs. The number of CPUs, GPUs and the memory allocation varied by model, depending on resource availability and scheduling within the cluster.

2.4 Approach, Analysis & Results

This section answers the four research questions and outlines the specific methodologies used to investigate each one.

2.4.1 RQ1: Performance of zero-shot LLMs using code diff and commit message

Both the commit message and the corresponding code diff contain information about the nature of the code changes. While the diff captures the structural modifications to the source code, the commit message provides a natural language description of the developer’s intent. LLMs offer a powerful mechanism to detect semantic signals indicative of bug-fixing behavior across these modalities. By leveraging their ability to reason over structured and unstructured data (code + natural language) [167, 44], LLMs offer a promising alternative to traditional rule-based or static analysis techniques. Furthermore, using prompt-based inference avoids the need for task-specific fine-tuning, making the approach adaptable across repositories and languages. However, the effectiveness of LLMs is sensitive to input length, as inference cost increases with the number of tokens in the prompt. Given that both code diffs and commit messages can vary significantly in length, it becomes essential to evaluate whether both are necessary for effective detection or whether the code diff alone suffices. This research question seeks to empirically determine whether commit messages provide additive value beyond the code diff for detecting tangled changes.

We investigate two scenarios using LLMs by designing tailored prompting strate-

gies for the binary classification of method-level diffs. Since no examples are provided within the prompt, this setup qualifies as zero-shot prompting.

Diff-only. The LLM is provided only with the code diff and instructed to assess whether the change represents a bug fix (*Buggy*) or not (*NotBuggy*). The prompt is adapted from Figure 2.1, with all references to commit messages and few-shot examples removed, and the instruction modified to require a single-word response.

Diff+Message. The prompt includes both the code diff and its commit message, instructing the model to assess their semantic alignment to identify bug-fixing intent. This prompt follows the *Diff-only Detection* setting, except that all references to commit messages are retained as in Figure 2.1.

Table 2.1 compares the performance of the evaluated LLMs under two input settings: *Diff-only* and *Diff+Message*. In the *Diff-only* scenario, *GPT-5* achieves the best overall performance, yielding the highest F1-score (0.767) and MCC (0.490). When commit messages are included, all models show substantial improvements. The *GPT-5* again attains the highest F1-score (0.879), while *GPT-4o* provides the best accuracy (0.874), precision (0.868) and MCC (0.747). Although *GPT-5* achieves the top F1-score, its high precision–recall gap indicates less balanced performance, whereas *GPT-4o* offers the most consistent results across all metrics.

The *Gemini-2.0-Flash* and the *GPT-OSS* models achieve high recall (> 0.9) but low precision (< 0.76) resulting in reduced F1-scores. The *GPT-OSS* models perform remarkably well in *Diff+Message* scenario despite their smaller size narrowing the gap with proprietary models (within 3–4% of *GPT-5*’s F1-score), likely benefiting from their unique dual-channel chat format [3], where an internal “analysis” channel

Table 2.1: Performance comparison of LLMs for Diff-only and Diff+Message. Metrics are calculated by targeting the buggy samples (pos_label = “Buggy”). For each scenario, the best value for each metric is highlighted in bold.

Scenario	LLM	Accuracy	Precision	Recall	F1-score	MCC
Diff-only	GPT-5	0.742	0.709	0.834	0.767	0.490
	GPT-4o-mini	0.704	0.660	0.860	0.747	0.426
	GPT-4o	0.691	0.654	0.831	0.732	0.395
	Gemini-2.0-Flash	0.692	0.650	0.853	0.738	0.402
	GPT-OSS-20B	0.663	0.615	0.900	0.730	0.364
	GPT-OSS-120B	0.667	0.619	0.900	0.733	0.372
Diff + Message	GPT-5	0.871	0.839	0.924	0.879	0.746
	GPT-4o-mini	0.822	0.764	0.941	0.843	0.661
	GPT-4o	0.874	0.868	0.886	0.877	0.747
	Gemini-2.0-Flash	0.796	0.717	0.988	0.831	0.639
	GPT-OSS-20B	0.820	0.758	0.950	0.843	0.662
	GPT-OSS-120B	0.827	0.766	0.948	0.847	0.671

generates intermediate reasoning (chain-of-thought) before producing the answer on “final” channel. Overall, these results demonstrate that integrating commit messages with diffs significantly improves performance, and that open-source LLMs are also effective despite their smaller sizes.

Summary of RQ1: Both proprietary and open-source LLMs can effectively detect tangled code changes using code diffs, with or without commit messages, though combining both inputs yields better results than code diffs alone.

2.4.2 RQ2: Performance of different prompting techniques

The effectiveness of LLMs highly depends on the input prompt, and the success of different prompting techniques is proven in various studies [235, 31, 236]. This research question investigates how different prompting techniques, namely *few-shot learning*, *chain-of-thought prompting*, and a hybrid approach combining both, impact the classification performance of LLMs.

Few-shot. The prompt includes labeled examples of buggy and non-buggy changes to guide the model, while the instruction remains identical to the *Diff+Message* setting of RQ1.

Chain-of-thought. The model is instructed to explain its reasoning in the output. The exact instruction prompt is shown in Figure 2.1, and no examples are included.

Chain-of-thought + Few-shot. This setting combines the above two techniques by including labeled examples and requiring the model to explain its reasoning in the output.

Based on the findings from RQ1, we use the combination of commit message and code diff for RQ2. Additionally, we exclude *GPT-4o-mini* from further experiments, as *GPT-4o* significantly outperformed this small model. This decision also supports cost efficiency, as the prompting techniques used in RQ2 involve substantial token consumption in both input and output.

Table 2.2: Performance comparison of different prompting techniques. Metrics are calculated by targeting the buggy samples (pos_label = “Buggy”).

Prompting Technique	LLM	Accuracy	Precision	Recall	F1-score	MCC
Few-shot	GPT-5	0.876	0.842	0.931	0.884	0.757
	GPT-4o	0.870	0.903	0.833	0.866	0.742
	Gemini-2.0-Flash	0.802	0.723	0.989	0.835	0.649
	GPT-OSS-20B	0.837	0.775	0.958	0.857	0.694
	GPT-OSS-120B	0.819	0.753	0.959	0.843	0.663
Chain of thought	GPT-5	0.863	0.829	0.920	0.872	0.730
	GPT-4o	0.827	0.770	0.940	0.846	0.669
	Gemini-2.0-Flash	0.765	0.688	0.984	0.810	0.587
	GPT-OSS-20B	0.829	0.765	0.956	0.850	0.679
	GPT-OSS-120B	0.829	0.770	0.946	0.849	0.676
Chain of thought + Few-shot	GPT-5	0.863	0.829	0.921	0.873	0.731
	GPT-4o	0.880	0.871	0.896	0.883	0.760
	Gemini-2.0-Flash	0.807	0.731	0.979	0.837	0.651
	GPT-OSS-20B	0.844	0.795	0.932	0.858	0.697
	GPT-OSS-120B	0.816	0.752	0.952	0.840	0.655

Table 2.2 presents the performance of different prompting strategies across LLMs. While *GPT-4o* with few-shot achieves the highest precision (0.903) but lower recall

(0.833), other models exhibit the opposite trend. In this setting, *GPT-5* attains the best overall performance with an F1-score of 0.884. For chain-of-thought prompting, performance generally decreases across models, though *GPT-5* again achieves the highest F1-score (0.872). When combining few-shot and chain-of-thought prompting, *GPT-4o* delivers the best results across all metrics except recall, which is highest for *Gemini-2.0-Flash*.

Among all prompting strategies, *GPT-5* with few-shot yields the top F1-score, while *GPT-4o* attains the highest accuracy, precision, and MCC with chain-of-thought + few-shot. Considering balanced performance across metrics, particularly the small gap between precision and recall (0.025), *GPT-4o* with chain-of-thought + few-shot offers the most balanced results, with an F1-score (0.883) nearly identical to the best (0.884). Open-source models also benefit consistently, though modestly, from advanced prompting; notably, *GPT-OSS-20B* remains competitive, often matching or surpassing *GPT-OSS-120B*.

A comparison between Table 2.1 and Table 2.2 reveals that few-shot prompting improves performance primarily for *GPT-5* and slightly for *GPT-OSS-20B*, while other models show marginal or decreased performance. Notably, chain-of-thought alone results in poorer performance across most models. However, combining few-shot prompting with chain-of-thought yields the most balanced and consistent outcomes overall, as observed with *GPT-4o*. This trend aligns with prior research suggesting that few-shot prompting can lead models to overfit to example formats rather than the task itself, while zero-shot chain-of-thought prompting may amplify biases and degrade reasoning quality [128, 204]. Our findings thus corroborate earlier studies

indicating that integrating few-shot and chain-of-thought generally outperforms zero-shot approaches [128].

Table 2.3: Predictions on a semantically misleading commit using chain-of-thought + few-shot prompting with *GPT-4o*.

Commit Message	Diff (Excerpt)	Ground Truth	Prompting Method	Prediction
Fixed bug 1050173 – <i>ImmutableFieldRule</i> no longer reports false positives for static fields. Also fixed version number in PMD.java.	<pre> ... - NameOccurrence occurrence = ... - if (occu- rance.isOnLeftHandSide()) { ... + NameOccurrence occ = ... + if (occ.isOnLeftHandSide()) { ... </pre>	NotBuggy	Commit message + diff (zero-shot)	Buggy
			Few-shot	Buggy
			Chain-of-thought	NotBuggy
			Chain-of-thought + Few-shot	NotBuggy

Case analysis: an example of the effectiveness of chain-of-thought approach in a semantically ambiguous commit

To better understand why the chain-of-thought technique improves model performance in the few-shot setting with *GPT-4o*, we analyze a representative example in-

volving semantic ambiguity between the commit message and the code modifications from the *PMD*⁴ project. As shown in Table 2.3, the commit message clearly describes a bug fix involving two files, *PMD.java* and *ImmutableFieldRule.java*. In this commit⁵, multiple methods are modified, including *initializedInConstructor()* from *ImmutableFieldRule.java*, which only includes variable renaming (e.g., *occurance* to *occ*) without affecting the program’s logic or behavior. No structural fix is present that addresses the reported issue. Despite this, both the zero-shot and few-shot models incorrectly classify the change as *Buggy*, influenced by the commit message. In contrast, the chain-of-thought technique, with or without few-shot prompting, correctly identifies the change as *NotBuggy*, consistent with the manually assigned label.

The reasoning produced by the chain-of-thought prompt illustrates a more grounded assessment. An important part of the reasoning states: “. . . However, in the provided diff, the only change is renaming the variable *occurance* to *occ*, which is a simple refactor for readability or consistency purposes. This modification itself doesn’t address the logic or behavior of the code in terms of resolving the stated bug. The commit message also mentions a version number fix in *PMD.java*, but no relevant changes are shown in this diff. . . .”

This case highlights the effectiveness of chain-of-thought in systematically analyzing complex commit messages that reference multiple issues. While such messages offer valuable context, they can also introduce ambiguity. The step-by-step reasoning decomposed the message, independently assessed each part against the code diff, and correctly identified the lack of structural or semantic alignment, ultimately concluding

⁴<https://github.com/pmd/pmd>

⁵<https://github.com/pmd/pmd/commit/a405d23dfb9e574e2b2ef23f1f45d548a738ed3b>

that the change was not a bug fix.

Summary of RQ2: Combining chain-of-thought reasoning with few-shot prompting yields the most balanced and consistent performance across models. While few-shot prompting alone benefits mainly *GPT-5*, and chain-of-thought alone often reduces accuracy, their combination enhances reasoning stability and mitigates overfitting. This hybrid approach also improves interpretability, enabling models to correctly handle semantically misleading commits through more context-aware reasoning.

2.4.3 RQ3: Performance of embedding-based machine learning models

This research question explores the viability of embedding-based ML models for detecting tangled code changes. Unlike prompt-based approaches that rely on in-context reasoning of LLMs, this method uses embeddings, vector representations of the input data from commit messages and code diffs, to train supervised classifiers. The key objective is to assess whether these embeddings, when used as features in ML models, can achieve competitive performance in detecting bug-related code changes. This experiment is motivated by the previous studies where embedding-based models performed well and even outperformed prompt-based techniques [186, 121].

The methodology employed for this research question diverges from that of the preceding research questions and the process illustrated in Figure 2.1. Specifically, this experiment omits the complex prompt engineering. Instead, only the core query, comprising the code diff and commit message, is retained and directly input to the

LLM to generate fixed-length vector embeddings for downstream classification. The modified methodology is shown in Figure 2.3.

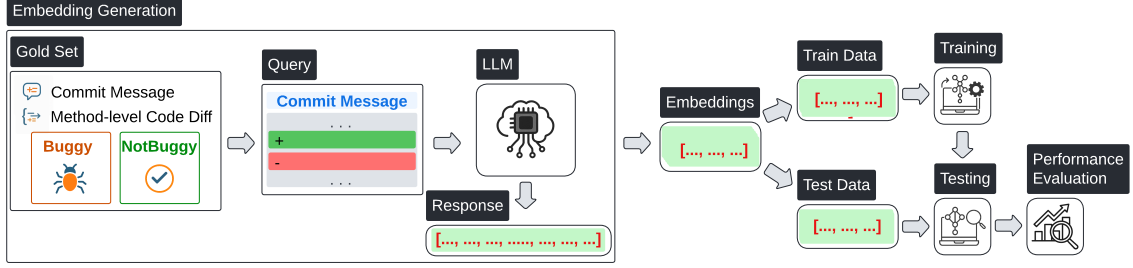


Figure 2.3: The methodology for RQ3 has two main parts: (i) embedding generation and (ii) building ML models.

To generate embeddings, Gemini’s *text-embedding-004* was employed with the *classification* task type. Due to resource constraints, this study focuses exclusively on this proprietary model, utilizing Gemini’s free quota. For the open-source models, a sliding-window approach with mean pooling was applied to address token input limits, 2048 for *EmbeddingGemma-300m* and 512 for *CodeBERT* and *GraphCodeBERT*. All models produced 768-dimensional vector representations for each instance, derived from the concatenation of the code diff and its corresponding commit message.

The classification task was treated as a binary problem, and the dataset was split into an 80% training set and a 20% test set. We evaluated multiple classifiers, including RandomForest (RF), Support Vector Classifier (SVC), DecisionTree (DT), and Multi-Layer Perceptron (MLP). These models were selected to represent a diverse range of learning paradigms: tree-based ensembles (RF, DT), margin-based optimization (SVC), and neural networks (MLP), which have been widely applied in various software engineering tasks [182, 51].

Table 2.4: Performance comparison of ML models trained on *text-embedding-004* embeddings. Metrics are calculated by targeting the buggy samples (pos.label = “Buggy”).

Classifier	Accuracy	Precision	Recall	F1-score	MCC
RF	0.816	0.841	0.82	0.83	0.629
SVC	0.884	0.918	0.866	0.891	0.768
DT	0.686	0.717	0.706	0.712	0.366
MLP	0.895	0.929	0.876	0.902	0.791

Table 2.4 presents the performance comparison of all models trained on *text-embedding-004* embeddings, showing that the *MLPClassifier* achieved the best results, with an accuracy of 0.895, F1-score of 0.902, and MCC of 0.791, outperforming other classifiers in all metrics. Similar experiments were conducted using open-source embeddings; however, due to space limitations, only the MLP results are reported in Table 2.5, as it consistently outperformed other classifiers. Among the open-source models, *GraphCodeBERT* combined with MLP achieved the highest performance (F1-score = 0.831), exhibiting a modest gap of around 7% compared to the proprietary model. This suggests that while proprietary embeddings retain an advantage, open-source alternatives offer competitive and cost-effective performance for this task.

Table 2.5: Performance comparison of MLP classifier trained on different LLM embeddings. The first row of *text-embedding-004* is taken from Table 2.4 for comparison. Metrics are calculated by targeting the buggy samples (pos_label = “Buggy”).

Model	Accuracy	Precision	Recall	F1-score	MCC
text-embedding-004	0.895	0.929	0.876	0.902	0.791
EmbeddingGemma	0.80	0.815	0.82	0.818	0.593
CodeBERT	0.765	0.738	0.833	0.782	0.533
GraphCodeBERT	0.828	0.831	0.83	0.831	0.656

Furthermore, we evaluated the `MLPClassifier` using the *text-embedding-004* embeddings under a Leave-One-Out (LOO) strategy to test its robustness, where a single example is used as the test sample and the remaining serve as the training data. This approach not only offers more robust evaluation, but also can show if a model’s accuracy can be improved with more training samples. Encouragingly, the model achieved improved performance with an accuracy of 0.9036, precision of 0.8980, recall of 0.9141, F1-score of 0.906, and MCC of 0.8073, setting a new benchmark compared to prior best results in RQ1 and RQ2. In terms of class-wise performance the model maintained a balanced detection capability, with precision and recall exceeding 0.89 for both classes and achieving F1-scores of 0.91 for *Buggy* and 0.9 for *NotBuggy*.

Summary of RQ3: Embedding-based ML models, particularly MLPClassifier, demonstrate strong predictive power for tangled change detection, achieving over 90% F1-score and outperforming prior baselines of RQ1 and RQ2. More encouragingly, our results suggest that the performance of embedding-based models can be improved with an enlarged training dataset.

2.4.4 RQ4: Potential impact of LLM-based untangling on future method-level bug prediction models

From the results of RQ1 to RQ3, we observe that the LLM-based method-level tangled change detection approach performs well with high accuracy and F1-scores using both prompting techniques and embedding-based ML models. This suggests that such approaches can effectively reduce noise in bug datasets. However, an important question remains: *Does this noise reduction benefit machine learning-based bug prediction models?* Building such a model, however, requires addressing several challenges. First, a broad set of code metrics is needed, whereas the dataset from Chowdhury et al. [51] includes only five. Second, Mashhadi et al. [150] demonstrate that combining code metrics with embeddings leads to better performance, requiring us to generate embeddings for all methods. Finally, a thorough evaluation across multiple algorithms and fine-tuning strategies is essential. Due to these requirements, constructing a full bug prediction model is beyond the scope of this chapter. Nonetheless, we aim to provide evidence that our untangling-based noise reduction yields a dataset that is more suitable for downstream method-level bug prediction. Therefore, this research question examines whether the metric distributions exhibit

stronger separation in our Less-Noisy dataset than in the original Noisy dataset. Improved separability does not, by itself, prove improved predictive performance; rather, it provides empirical support that noise reduction can strengthen the signal available to future bug prediction models.

In RQ1 and RQ2, we evaluated multiple prompting strategies and LLMs, showing that *GPT-4o* combined with chain-of-thought + few-shot prompting delivers the best and balanced performance comparing all metrics. In RQ3, we examined the feasibility of using LLM-generated embeddings with downstream machine learning classifiers to detect tangled changes. Although the latter approach yields better results, it requires generating embeddings for all tangled code diffs from bug-fix commits to support prediction with the machine learning model, which is more complex and time-consuming than the chain-of-thought + few-shot approach. Therefore, we employed *GPT-4o* with chain-of-thought + few-shot prompting to detect tangled changes across all methods in 49 software projects in a fast and efficient manner.

In the dataset by Chowdhury et al. [51], each method is associated with its change history and the number of other methods modified in the same commit. We apply the few-shot and chain-of-thought strategy using *GPT-4o* to separate *NotBuggy* changes within bug-fix commits, thereby removing noise. This process does not introduce new instances; rather, it modifies the dataset by removing *NotBuggy* methods from bug-fix commits and labeling them as *NotBuggy*. To enable replication, the algorithm for creating the Less-Noisy dataset is included in the shared repository (algorithm/algorithm.png). After constructing the *Less-Noisy* dataset, we obtain four types of methods:

Noisy NotBuggy. Following Chowdhury et al. [51], methods are labeled as *NotBuggy* if they are at least two years old and never involved in a bug-fix. This set is called *Noisy* because it comes from the original noisy dataset, which excludes non-buggy methods that were mixed with buggy changes.

Less-Noisy NotBuggy. It expands on the previous set by including methods confirmed as *NotBuggy* after LLM-based untangling.

Noisy Buggy. Following Chowdhury et al. [51], methods are labeled as *Buggy* if they appear in bug-fix commits, without accounting for tangling, hence the dataset is considered *Noisy*.

Less-Noisy Buggy. This set includes methods identified as truly buggy after applying our LLM-based untangling approach.

The dataset provided by Chowdhury et al. [51] contains five code metrics computed for each version of a method across its history. For our analysis, we consider the metrics from the first version of each method to ensure consistency across the dataset. These code metrics are defined as follows: *Size* refers to the number of source lines of code, excluding comments and blank lines [53, 51]; *Readability* is a score that reflects the ease of reading the source code, developed by Buse et al. [33]; *McCabe* measures cyclomatic complexity by counting the number of independent execution paths in a method [153]; *FanOut* represents the number of distinct methods called by a given method, indicating its dependency footprint [182, 51]; and *Maintainability Index (MI)* is a composite metric that combines several aspects, including complexity and size, to offer a comprehensive assessment of code maintainability [173].

Figure 2.4 presents the distribution of these 5 code metrics across the four method

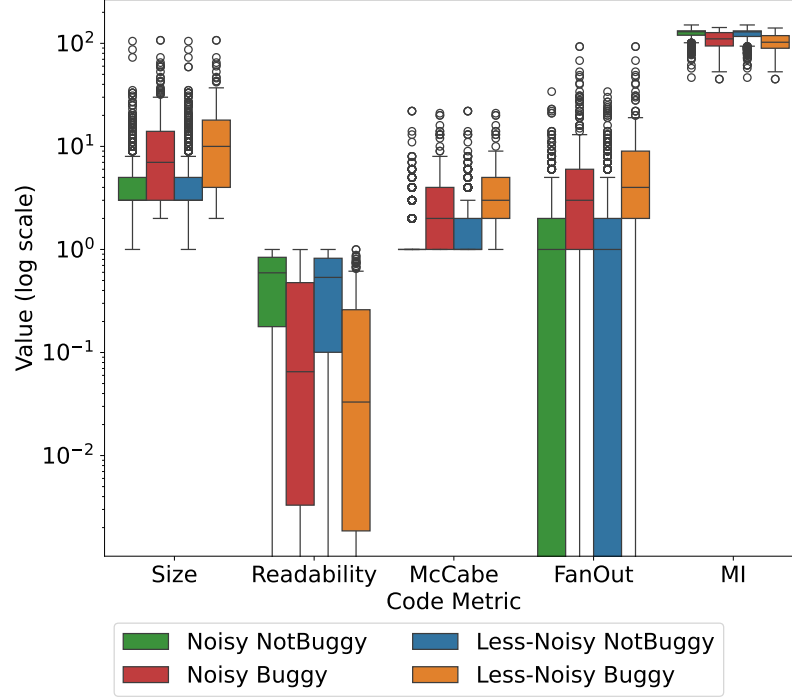


Figure 2.4: Distribution of code metrics for *Buggy* and *NotBuggy* methods in the *Titan* project across the *Noisy* and *Less-Noisy* datasets. Metric distributions differ more significantly in the *Less-Noisy* dataset.

sets for the project *Titan*. Visually, the difference in *Size* between *Less-Noisy Buggy* and *Less-Noisy NotBuggy* methods appears larger than the other two sets. A similar pattern is observed for all other code metrics. To determine whether these differences are statistically significant, we use the Wilcoxon rank-sum test [154]. This non-parametric test is widely adopted in software engineering research due to its robustness in comparing distributions [19, 43, 49]. The results show that the differences in distributions are indeed statistically significant ($p \leq 0.05$) for all code metrics.

As the differences are statistically significant for both datasets, we calculated Cliff’s delta effect size [147] to understand the magnitude of these differences. This

non-parametric measure is also widely used in software engineering research [19, 43, 49] and does not require assumptions about the data distribution. The results show that, in the *Noisy* dataset, the effect size is large for *MI* and medium for all other code metrics. In the *Less-Noisy* dataset, the effect size is large across all code metrics.

To evaluate the generalizability of our results, we performed Wilcoxon rank-sum tests across all 49 projects. In the *Noisy* dataset, significant differences ($p \leq 0.05$) were observed in 47 projects (95.92%) for *Size*, *McCabe*, and *MI*, 45 projects (91.84%) for *Readability*, and 48 projects (97.96%) for *FanOut*. In the *Less-Noisy* dataset, we observe improved results, with one additional project exhibiting statistically significant separation for each metric.

To further quantify the differences between *Buggy* and *NotBuggy* methods, we compute Cliff’s Delta effect sizes across all 49 projects. Table 2.6 presents the effect size distributions for both the *Noisy* and *Less-Noisy* datasets. We categorize the effect sizes following Hess et al. [105]: *Negligible* (< 0.147), *Small* ($0.147 \leq \delta < 0.33$), *Medium* ($0.33 \leq \delta < 0.474$), and *Large* ($\delta \geq 0.474$).

The table reports the proportion of projects falling into each effect size category for all five code metrics. The results indicate that, for most metrics, the *Less-Noisy* dataset shows a substantial shift toward larger effect sizes. For example, for the code metric, *Size*, in the *Less-Noisy* dataset, 73% of the projects exhibit a large effect size in the difference between *Buggy* and *NotBuggy* methods, compared to only 36% in the *Noisy* dataset. Similar patterns are observed for the other four code metrics. These findings clearly demonstrate that our filtering approach effectively eliminates noise and increases the distinction between the code metrics of buggy and non-buggy

Table 2.6: Cliff’s Delta effect size values compare *Noisy* and *Less-Noisy* datasets, grouped by effect size category and shown as percentages in the format *Noisy* / *Less-Noisy*. The percentages for the *Noisy* dataset do not sum to 100% because the *RxJava* project was excluded as it had no *NotBuggy* methods.

Metric	Negligible	Small	Medium	Large
Size	0 / 0	14.29 / 6.12	46.94 / 20.41	36.73 / 73.47
Readability	2.04 / 2.04	36.73 / 24.49	42.86 / 40.82	16.33 / 32.65
McCabe	2.04 / 2.04	34.69 / 8.16	36.73 / 42.86	24.49 / 46.94
FanOut	0 / 0	10.20 / 6.12	46.94 / 22.45	40.82 / 71.43
MI	0 / 0	12.24 / 6.12	34.69 / 16.33	51.02 / 77.55

methods. As a result of this enhanced separability, we believe that our approach will support improved performance in future machine learning models for bug prediction.

Summary of RQ4: After constructing the *Less-Noisy* dataset using the LLM-based approach, statistical analysis reveals significantly improved separability (with larger effect sizes) in code metrics between buggy and non-buggy methods. This suggests that the noise reduction could enhance the performance of future ML-based bug prediction models.

2.5 Threats to Validity

Construct Validity. The manual annotation of code diffs as *Buggy* or *NotBuggy* may be affected by subjective bias. Although annotators were provided with explicit guidelines and a UI tool to assist in their evaluations, subjective judgment and

semantic ambiguity in commit messages may have influenced labeling outcomes.

External Validity. The dataset used in this study comprises only open-source Java projects, which may limit the generalizability of our findings. Applying this approach to other programming languages, project domains, or commit conventions may require modifications to the prompts or retraining of models.

Internal Validity. The dataset used in this study was curated from method histories of Java projects obtained through CodeShovel [89], which is not fully accurate. Any inaccuracies in the method histories could impact the results.

Conclusion Validity. It is affected by all of the above mentioned threats.

2.6 Conclusion and Future Work

Method-level bug prediction is considered one of the holy grails in software engineering research. However, it remains an open challenge, primarily due to the absence of a noise-free method-level bug dataset [51, 182]. To address this, we evaluated the effectiveness of various LLMs and prompting strategies for detecting tangled changes at the method level. Our results show that even zero-shot LLMs achieve high accuracy when both commit messages and code diffs are provided (RQ1). Performance improves further when combining chain-of-thought with few-shot prompting (RQ2). Additionally, embedding-based models deliver even greater accuracy gains (RQ3). Across RQ1 to RQ3, open-source models also demonstrate strong performance. While they do not yet match proprietary models, they offer efficient alternatives with acceptable trade-offs. Building on these insights, we generated a *Less-Noisy* dataset using our LLM-based method. This dataset shows promise in developing more accu-

rate method-level bug prediction models as it exhibits a significantly stronger power of commonly used code metrics to differentiate between bug-prone and non-bug-prone methods compared to the original *Noisy* dataset (RQ4).

The implications of our findings extend beyond the untangling problem and contribute to broader discussions on the capabilities and limitations of LLMs in software engineering. Although challenges such as hallucinations, reproducibility, and data leakage remain [239, 113, 240, 197], our study offers empirical evidence supporting the effectiveness of LLMs in tasks such as code reasoning, classification, and dataset construction [253, 112, 167]. In particular, our work shows that when guided by carefully designed prompts, LLMs can significantly improve software engineering workflows.

Looking ahead, a promising direction for future research is to enrich the dataset with additional code metrics and embeddings and to explore a range of machine learning models for building more effective method-level bug prediction models. To support continued progress, we have publicly released our code and dataset, allowing researchers to use them directly or adapt them to their specific needs.

Chapter 3

Method-Level Bug Prediction: How Far Are We?

Abstract

Software bugs are costly to locate and repair, making method-level bug prediction (MLBP) an attractive means of focusing developer effort on the methods most likely to be defective. Progress in MLBP has been held back by three persistent challenges: noisy bug labels, unrealistic evaluation protocols, and reliance on hand-crafted software metrics. We revisit MLBP at scale across 100 open-source Java projects by addressing all three challenges within a unified pipeline. We clean labels through LLM-based untangling, producing *Noisy* and *Less-Noisy* dataset variants; enrich metric-based features with source-code embeddings; and evaluate under two realistic settings, *Inception* for predicting defects at a method’s introduction and *Evolution* for predicting bug-proneness as the method evolves.

Cleaner labels significantly improve inception-time prediction, lifting MCC from 0.13 to 0.16, with smaller but consistent gains in the evolution setting. Fused embeddings push validation MCC and buggy-class F1 to 0.26 and 0.29 at inception, but the gains do not survive testing. Severe class imbalance and weak cross-project generalization cut the best inception model to just 0.18 MCC, and in the evolution setting, embeddings actually underperform the tabular baseline (0.12 vs. 0.15). Cleaner labels and richer features help, but are not enough on their own. Therefore, closing the remaining gap will likely need learning methods built for extreme imbalance and per-project adaptation.

3.1 Introduction

Software maintenance is the most expensive and longest-running phase of the software development lifecycle, often exceeding the initial development cost [69]. Bug detection and resolution alone consume 50–70% of the total cost [233], and bugs were estimated to cause \$1.7 trillion in losses in a single year [201]. Undetected bugs degrade user experience [23], and can be life-threatening in safety-critical systems [54]. Because bugs are cheaper and safer to remove early in the lifecycle [51], predicting them has long been a classic, heavily studied problem and is central enough to the field to be called the “prince of empirical software engineering research” [131].

Bug prediction has been studied at several granularities: commit, file, class, method, and line [257, 11, 21, 24, 83, 219, 241]. Coarse granularities are of limited practical use because a single buggy commit, file, or class, spanning thousands of lines, leaves little for a developer to act on [51, 257]. Commit-level, just-in-time prediction is

the established alternative [119], yet commits are only partially defective: almost 43% mix buggy and clean files [181]. Line-level prediction is fine-grained but noisy, since a line’s bugginess often depends on its surrounding context [187, 234]. Method-level bug prediction (MLBP) occupies the practical middle ground. A method is coarse enough to be self-contained yet fine enough to localize a fault [81], and developers prefer it most when acting on predictions [228, 51].

MLBP has a long research history, with studies reporting high accuracy [75, 210, 161], but this success did not survive rigorous evaluation. Earlier models [75, 81, 98, 158, 161, 210] were evaluated with time-insensitive k-fold cross-validation. Pascarella et al. [182] reproduced the work of Giger et al. [81] and found it to fail under realistic evaluation. Chowdhury et al. revisited this result with more datasets and evaluation scenarios and reached the same conclusion [51]. Together, these results establish method-level bug prediction as an open problem.

This failure has three well-documented causes. *Inaccurate labeling* is the most prominent: datasets built from commit-message keywords mislabel much of the data [194, 51, 28], and labels are further corrupted by *tangled changes*, bug-fix commits that bundle unrelated edits so incidentally touched methods are still labeled buggy. Tangled changes account for an estimated 7–20% of bug fixes and can mislabel more than 16.6% files [103, 104], the single largest source of label noise. *Unrealistic evaluation* compounds the problem, since training on information unavailable at prediction time inflates accuracy [114, 19]. Finally, prior models rely almost entirely on *shallow hand-crafted features* [81, 182, 75, 98, 158, 210], leaving the code’s semantic content untapped [161]. This limitation also represents an opportunity. Transformer-based code

representations capture semantic information that hand-crafted metrics miss [72], and learned features outperform hand-crafted ones for defect prediction [231]. However, they have not yet been applied to method-level bug prediction or combined with method-level metrics, developer, and bug-history features.

Motivated by these key limitations, we ask how well method-level bug prediction performs when addressed together. We conduct, to the best of our knowledge, the first large-scale study to confront all these challenges within a single method-level pipeline. Our contribution is threefold. First, we construct higher-quality datasets by following the labeling scheme of Chowdhury et al. [51] with the LLM-based untangling approach presented in the previous chapter (Chapter 2). Second, we evaluate two realistic prediction settings: *Inception*, which predicts a method’s eventual bug-proneness using only information available at its introduction, and *Evolution*, which predicts bug-proneness across method versions using both project-wise and temporal splits. Third, we incorporate semantic code embeddings (Qwen3-Embedding-8B) and fuse them with tabular features in a two-tower network. Our study spans 100 open-source Java systems with code, change and bug history, developer metrics, and embedding data, and is organized around two research questions, one for each dataset family.

RQ1. Can we predict whether a method is buggy at its inception? We predict whether a method will ever become buggy using only features available when it is introduced. Untangling improves MCC from 0.13 to 0.16 and F1-Buggy by 0.02, while adding fused code embeddings further increases MCC to 0.18 and F1-Buggy by 0.06. However, with only 2.56% Buggy methods in the test set, an MCC of 0.18 remains too low for practical use, indicating that inception-time bug prediction is still

an open problem.

RQ2. Can we predict whether a method is buggy at each stage of its evolution? We predict the bug status of every method version over time. Untangling provides modest gain, but code embeddings do not outperform the tabular baseline (MCC 0.12 vs. 0.15). In the project-disjoint setting, the best MCC is only 0.15, whereas a temporal split improves it to 0.224. Still, it evaluates an easier within-project prediction scenario rather than generalization to unseen projects.

Across both settings, we address the three main challenges of method-level bug prediction—inaccurate labeling, unrealistic evaluation, and shallow features—yet the problem remains unsolved. Our results indicate that severe class imbalance and poor cross-project generalization are the primary bottlenecks. Establishing this ceiling with a realistic, embedding-augmented pipeline is itself a contribution. The remaining gap is unlikely to be closed by cleaner labels or richer features alone; instead, future work should focus on imbalance-aware learning and per-project adaptation. We share our datasets and code publicly to support future research¹.

3.2 Related Work and Motivation

In this section, we discuss prior studies on method-level bug prediction and how they motivated this research.

¹<https://github.com/SQMLab/MLBP>

3.2.1 Method-Level Bug Prediction and Prediction Granularity

Bug prediction has historically operated at the file or module level [11, 21, 24, 83, 165, 255, 257], which is too expensive to act on, leaving developers searching for the offending code [182, 194]. A finer granularity, such as line-level prediction, is a natural alternative [118], but they suffer because many lines look alike by chance [218, 202], and a line’s bugginess often depends on its surrounding context [187, 234]. Method-level prediction sits between these extremes and has drawn wide attention [75, 81, 98, 158, 161, 210, 180, 51]. Giger et al. proposed the first *method-level* model and reported precision and recall above 80% on 21 Java systems [81]. Pascarella et al. reproduced these models and found accuracy held within a release but collapsed under cross-release evaluation [180], later sharpened into an explicit negative finding [182]. Subsequently, Chowdhury et al. [51] confirmed that labeling noise, concept drift, unrealistic evaluation, and project dissimilarity keep the task open. Most prior work scores a method once at a fixed point in its history, i.e., the *inception* view isolated in RQ1. Few treat bug-proneness as evolving rather than fixed. Wang et al.’s continuous bug-prediction setup [232], which trains on chronological versions, is the closest prior work to our Evolution family. A version-to-version GNN model [233] similarly captures dependencies across successive versions to reduce false positives. However, none combine version awareness, noise reduction, and realistic splitting—gaps addressed by our Evolution family.

3.2.2 Bug Labeling: Keyword Heuristics and Tangled Changes

Supervised bug prediction is only as reliable as its labels, and automated labeling is inherently noisy. Methods are typically marked buggy if modified in a bug-fix commit, identified via keyword matching in commit messages. The SZZ algorithm formalized this idea [215] and was refined by several variants [125, 61]. However, keyword sets vary across studies [83, 217, 194], introducing substantial noise [51, 28]. We adopt Chowdhury et al.’s improved scheme, which combines multiple keyword sources [51].

Beyond keyword noise, another major source is tangled changes, where bug fixes are mixed with unrelated edits. Herzig and Zeller reported that many fixes are tangled [103], and untangling them improves prediction [104]. Subsequent work proposed untangling methods ranging from commit-time detection [126], IDE-interaction methods [65], to program-dependence approaches [179]. More recently, in the previous chapter of this thesis (Chapter 2), we introduced an LLM-based method at the method level, improving dataset quality and metrics separability. We use this approach to construct our *Less-Noisy* dataset variants.

3.2.3 Code, Change, and Developer Metrics

Bug prediction models use source-code metrics [21, 75, 83, 166, 257, 51], historical change metrics [211, 163, 88, 47], and developer metrics [171, 209]. Code metrics quantify intrinsic complexity; e.g., McCabe’s cyclomatic complexity [153] counts independent paths through a method, reflecting testability and maintainability. The Chidamber and Kemerer suite extended this to object-oriented design [46] and has

been reported as a good predictor [21, 93]. While static metrics support learnable predictors [158], their limits remain debated [62], though this debate weakens at the method level [130, 50].

Change metrics measure how code evolves. Graves et al. found that change history often outperforms product metrics for fault prediction [88], supported by churn-based measures [165, 163]. Rahman et al. similarly observed that the predictive power of code metrics plateaus as a project evolves [192]. Developer metrics attribute risk to authorship; e.g., low ownership correlates with higher defect rates [29], while focused developers introduce fewer errors [171]. However, no single metric family consistently dominates [94, 190].

Recently, this debate has shifted to the method level. Chowdhury et al. show that code-metric distributions differ across method sizes and that unchanged methods are unlikely to change after two years [53], which motivates our conservative labeling window.

3.2.4 Learned Code Representations

A parallel line of work learns representations directly from source code. Aladics et al.’s Doc2Vec and Qu and Yin’s extended node2vec, both source code representations, outperformed traditional metrics-based models [6, 188]. Other studies incorporated auxiliary signals, e.g., Huo et al. introduced CAP-CNN [108] to embed comments with code, and Wang et al. showed that deep belief network features can outperform hand-crafted metrics [231]. More recently, pre-trained models like CodeBERT [72], and LLM-based representations have shown strong effectiveness to defect prediction [251].

Rather than replacing metrics, an emerging trend is to fuse them with learned embeddings. Ni et al. combined change metrics with CodeBERT embeddings for just-in-time defect prediction [169], and recent work explores gated or hybrid fusion of semantic and structural features [2, 156]. However, such fusion has not yet been tested at the method level, and we address this through a two-tower fusion classifier.

3.2.5 Realistic and Temporal Evaluation

Evaluation design strongly affects whether reported accuracy holds in practice. Random k-fold cross-validation, widely used in prior work [257, 11, 21, 24, 83, 219, 241], can leak future information and inflate performance. Pascarella et al. showed that inter-release evaluation drops method-level performance to near random [182]. Chowdhury et al. likewise argued for time-sensitive evaluation as a precondition for credible method-level prediction [51], and recommended a “train on the past, predict the future” protocol or a project-wise split where all of a project’s data falls into a single set. Practitioner surveys also confirm that realistic evaluation is the key need [118, 228]. This motivates our project-wise split and the RQ2 version-wise temporal split.

Building on these gaps, we adopt the open-source Java corpus and labeling lineage of [51] and extend the LLM-based untangling approach of Chapter 2 to a dataset-scale setting. We study two underexplored research questions in method-level prediction: defect predictability at inception and across evolution. For both, we evaluate three methodological treatments: noise reduction, learned embeddings, and evaluation under both project-wise and temporal data splits. To the best of our knowledge, this is

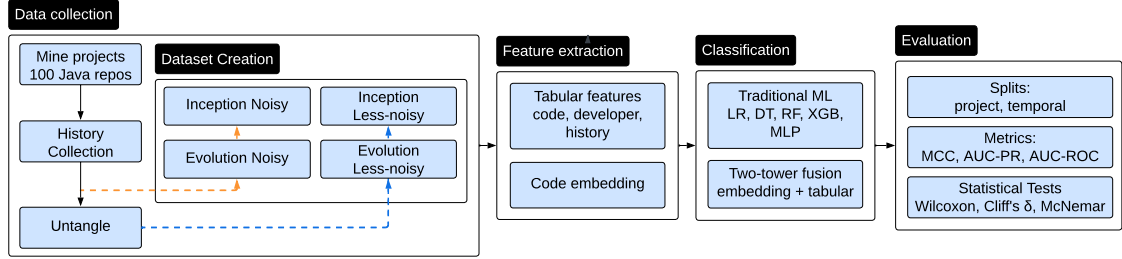


Figure 3.1: Method-level bug prediction pipeline. The pipeline comprises four phases: (1) data collection, which constructs four datasets (Inception/Evolution $\times$ Noisy/Less-Noisy); (2) feature extraction using software metrics and code embeddings; (3) classification with a two-tower fusion network and traditional baselines; and (4) evaluation under project and temporal splits using imbalance-aware metrics.

the first method-level study to combine all three in a unified framework.

3.3 Methodology

Figure 3.1 illustrates our pipeline, which moves from raw repository histories to trained bug-prediction models in four phases: data collection, feature extraction, classification, and evaluation. We describe each phase in detail below.

3.3.1 Data Collection

Project Selection

We build on the Java corpus of Chowdhury et al. [51], inheriting their 49 open-source projects and expanding it to 100 projects by mining additional repositories from GitHub. To collect new projects, we queried the GitHub REST API via the

PyGitHub library,² following established best practices for mining GitHub data [117]. We retrieved the 1,000 most-starred repositories and applied a series of filters. To reflect contemporary practice, we retained only repositories created within the last 15 years and active within the past year, excluding templates, archived repositories, and forks through the corresponding API parameters. To ensure a substantial and representative Java codebase, we further required at least 2,000 commits and 95% Java composition. Table 3.1 shows a sample of 10 projects and their properties. The full list is available in our replication package.

Table 3.1: Representative sample of studied subject systems spanning the range of popularity and size.

Project	# Stars	# Forks	# Commits	# Methods	Snapshot
spring-boot	79,791	41,839	59,538	45,247	f8f1da6
MPAndroidChart	38,212	9,054	2,072	1,914	9c7275a
SmartTube	27,814	1,493	12,906	20,480	46dd608
Hystrix	24,460	4,715	2,121	4,924	5ce3bc5
dagger	17,687	2,040	4,566	6,489	f9deeeef
jib	14,275	1,465	2,385	3,649	349c809
auto	10,547	1,212	2,052	2,620	8a2b2d0
seatunnel	9,080	2,163	5,298	21,399	3f1fd54
junit4	8,530	3,283	2,518	3,839	71c33ce
MinecraftForge	7,483	2,796	9,002	5,493	07f6499

²<https://github.com/PyGithub/PyGithub>, last accessed 28/06/2026

History Collection

For each project, we construct the version history of every method using HistoryFinder [111], which tracks a method across commits by string and structural similarity, correctly handling renames and moves. This produces an ordered list of commits per method, which is the basis for deriving labels and the change history of methods for our dataset. We restrict our analysis to source-code methods by dropping obvious test and demo methods located in the *test* or *example(s)* folder based on Java naming conventions.

Dataset Creation

Each method, or method version, is assigned one of two labels: *Buggy* or *Not-Buggy*. We produce four datasets along two orthogonal axes: two families that differ in prediction granularity (*Inception* and *Evolution*) and, within each family, two noise variants (*Noisy* and *Less-Noisy*). We first describe the base labeling rules shared by all four, then explain how the noise variants and families are derived.

Base Labeling Rules We follow the keyword-based labeling approach of Chowdhury et al. [51], who identified limitations in earlier keyword sets introducing noise and proposed a more accurate alternative. A commit is treated as a bug-fix commit when its message contains one or more keywords, such as *bug*, *fix*, or *error*. Every method touched by such a commit is assumed to carry a bug in its immediately preceding version and is labeled *Buggy*. Prior works typically label a method *NotBuggy* if it has never appeared in a bug-fix commit [81, 161]. Chowdhury et al. [51] argued against this, showing that recently introduced methods often have not had enough time for

defects to be discovered and fixed. We, therefore, adopt a conservative five-year observation window: a method is labeled *NotBuggy* only if it has not participated in any bug-fix commit and is at least five years old.

Noise Variants - *Noisy* and *Less-Noisy* Applying the keyword-based rule directly produces the *Noisy* dataset, in which every method touched in a bug-fix commit is labeled *Buggy*, even when the fix is tangled with unrelated modifications [103]. To address this, we apply the LLM-based untangling approach presented in Chapter 2 to re-evaluate whether each modified method actually participated in the bug fix. Given the high computational cost of LLM-based untangling, we use the open-weight *GPT-OSS-20B* model [3], which, as shown in Chapter 2, achieves performance comparable to proprietary LLMs at substantially lower cost. Relabeling with the untangling process produces the *Less-Noisy* dataset; because the process removes incorrectly marked *Buggy* instances, the *Less-Noisy* variant contains fewer *Buggy* samples than its *Noisy* counterpart.

Dataset Families - *Inception* and *Evolution* The two families apply the rules above at different granularities. In the **Inception** family, each method contributes its first version whose label reflects the method’s entire lifetime: *Buggy* if it was ever modified in a bug-fix commit and *NotBuggy* only if it was never touched in a bug-fix commit and is at least five years old. In the **Evolution** family, each version of a method contributes as a separate instance. A version is *Buggy* if the immediately following modification occurs in a bug-fix commit—meaning that version is what the fix corrected. However, a version is *NotBuggy* only if neither it nor any of its later

versions is involved in a bug-fix commit and at least five years of project history follow it. Figure 3.2 shows the resulting class distributions across all four datasets.

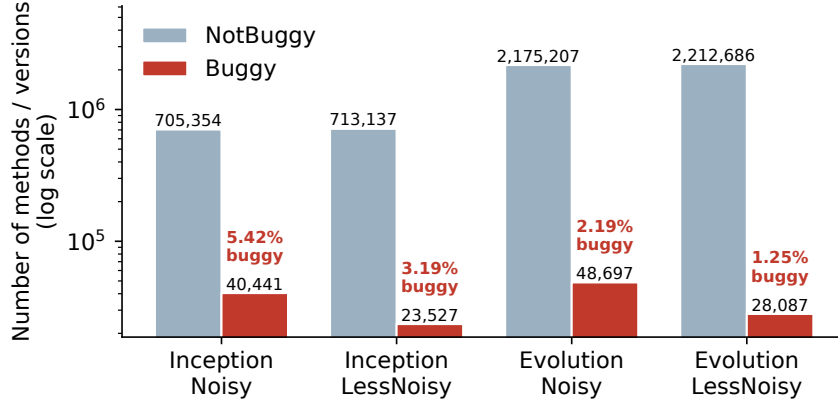


Figure 3.2: Number of NotBuggy and Buggy methods, and the percentage of buggy methods. We further resample all the dataset so that both variant in a family has same number of samples.

Since our experiments directly compare models trained on the *Noisy* and *Less-Noisy* variants, differences in class distribution could confound the results. To ensure a fair comparison, we perform project-wise sampling so that, for each project, the *Noisy* and *Less-Noisy* datasets contain identical numbers of *Buggy* and *NotBuggy* instances.

3.3.2 Feature Extraction

Each labeled instance, a method in the *Inception* family or a method version in the *Evolution* family, is represented by two complementary modalities: a block of *tabular* features that summarize the method numerically and a *code embedding* vector that captures its source-level semantics. Table 3.2 lists the feature sources and the family

each one is available to.

Table 3.2: Feature sources and their availability.

Source	Example features	Inception	Evolution
Code metrics	SLOC, McCabe, fan-out	✓	✓
Developer	#unique devs, contribution counts	✓	✓
Change history	edit distance, diff size, recency		✓
Bug history	time since last bug, last-fix distance		✓
Code embedding	Qwen3-Embedding-8B (4096-d)	✓	✓

Tabular Features

We extract four categories of features motivated by the prior studies [119, 209, 192, 47, 28].

Code metrics These features characterize each method independently of its change history. Following Chowdhury et al. [47], we extract structural code metrics, including SLOC, cyclomatic complexity (McCabe) [153], fan-out, and additional metrics such as McClure and unique fan-out. In total, we collect 39 code metrics, all of which are available for both dataset families.

Developer metrics These capture who has worked on the code or the project. For both families, we record the number of distinct developers who have contributed to the project and the contribution count of the developer responsible for the labeled

commit. The *Evolution* family additionally tracks the developer’s contribution counts on the method and the total number of contributors to that method. All counts are accumulated from the project’s oldest commit up to the labeled commit, ensuring no future activity leaks into the features.

Change-history metrics (Evolution only) These summarize how the method has changed over its lifetime and represent the process signal that has repeatedly shown to rival or outperform static code metrics for fault prediction [88, 192]. We extract 19 metrics, including the edit distance between consecutive versions, raw diff sizes, and the time elapsed since the previous revision. A method at inception has no such history, so these features exist only for the *Evolution* family.

Bug history metrics (Evolution only) These capture the method’s prior bug-fixing activity—14 metrics in total, including the time since its last bug-fix commit and the edit distance of that fix. Like the change-history metrics, they are defined only once a method has accrued a bug-fix history and are restricted to the *Evolution* family.

Code Embeddings

Hand-crafted metrics capture structure but say nothing about meaning. To complement them, we follow the line of work showing that learned representations of code add predictive signal beyond static metrics [231, 72]. Therefore, we create a single dense embedding of each method’s source code. Specifically, we encode the concatenation of a method’s documentation comment and its body using *Qwen3-*

Embedding-8B [250], a 4096-dimensional text-embedding model that ranks among the top models on the MTEB benchmark leaderboard³. We chose this model for two reasons. First, large language models pretrained on diverse natural language and programming language corpora learn richer contextual representations of source code than earlier encoder-only models. Second, Qwen3 substantially outperforms CodeBERT [250] on code-related benchmarks, suggesting it captures deeper syntactic and semantic properties of programs.

3.3.3 Classification Models

Every model shares a common preprocessing and training pipeline so that differences in reported performance reflect the model itself rather than incidental implementation choices. Tabular features are first standardized using *StandardScaler* [184]. We then address class imbalance (buggy methods are the minority class) through a resampling step that is itself an experimental factor. We evaluate four strategies: *i) no resampling*, where we train on the native class ratio; *ii) SMOTE*, which synthesizes new minority instances by interpolating between existing neighbours rather than copying them [40]; *iii) random over-sampling (ROS)*, which duplicates minority instances; and *iv) random under-sampling (RUS)*, which discards majority instances [134]. The pipeline ends in one of the classifiers described below.

³<https://huggingface.co/spaces/mteb/leaderboard>, last accessed 28/06/2026

Traditional Machine Learning (ML) Models

We evaluate five standard ML classifiers spanning linear, tree, ensemble, and neural families: *i)* *logistic regression*, as a linear baseline; *ii)* *decision tree*, a single recursively partitioned tree; *iii)* *random forest*, an ensemble of decision trees grown on bootstrap samples and feature subsets [30]; *iv)* *XGBoost*, a regularized, scalable realization of gradient-boosted trees [42]; and *v)* *multilayer perceptron (MLP)*, a feed-forward neural network. When no resampling is applied, each classifier handles class imbalance through its own built-in mechanism: balanced class weights for logistic regression and the decision tree, a balanced bootstrap subsample for the random forest, a positive-class weight for XGBoost, and early stopping on the validation split for the MLP [184, 42]. These models are used as baselines and only trained with tabular features.

Fusion Models

A fusion model is a deep learning technique that combine data from different sources [77]. Inspired by the previous studies [156, 139, 2, 229], we employ a fusion model in our case. Instead of simply flattening both modalities (tabular features and embeddings) into a single feature vector, the fusion model treats them explicitly through a two-tower architecture. One tower projects the high-dimensional code embedding down to a compact representation, and the second tower encodes the tabular features. A shared fused head then combines the two outputs to produce the final *Buggy/NotBuggy* prediction label.

We explore five fusion strategies for combining the two tower outputs: *i)* *Con-*

cat, concatenates the two representations and passes them through an MLP head, the standard late fusion baseline; *ii) Gated*, learns a sigmoid gate that dynamically reweights the two modalities before combining them, allowing the network to suppress whichever is less informative for a given instance [16]; *iii) Attention*, replaces the fixed gate with an attention mechanism that scores and weights each modality [225]; *iv) Feature-wise Linear Modulation (FiLM)*, lets the tabular tower modulate the code representation through a learned per-feature scale and shift, rather than simple concatenation [185]; and *v) Attention-Gated*, composes the attention and gating mechanisms. All five variants share the same tower architecture, head width, and optimizer, so the only difference between them is the fusion strategy itself.

For class imbalance, the fusion model also uses one of the four resampling strategies similar to the traditional models. When no resampling is used, we instead add a positive-class weight directly into the binary cross-entropy loss [183], which handles the class imbalance by making the model penalize mistakes on positive samples more heavily. All fusion models are trained with early stopping on the validation split.

3.3.4 Evaluation

The reported performance of a model is only meaningful if the evaluation reflects real use. We therefore evaluate every model on held-out data under a protocol matched to the research question it addresses, prioritizing metrics suited to class imbalance and testing differences non-parametrically rather than relying on a single aggregate score.

Dataset Splits

Project-level split. We randomly partition all the projects into training, validation, and test sets in a 70/10/20 ratio. Every method belonging to a given project, therefore, falls in a single split, and no method, or near-duplicate of it, crosses the boundary. We reuse the same project partition across both noise variants and both families for consistent comparisons.

Version-wise temporal split (Evolution only). For RQ2, where the task is to predict the bug-proneness of each version as the method evolves, a project-disjoint split ignores time. We therefore construct a temporal split: within each project, we order versions by commit time, train on the oldest $n\%$ of versions, and predict on the remainder, splitting newest versions into validation ($\frac{1}{3}$) and test ($\frac{2}{3}$). We vary n over 30%, 50%, and 70% and produce three variants. This “train-on-the-past” design is the field standard for time-aware evaluation [19, 51] and avoids the leakage introduced by a random split.

Evaluation Metrics

We report standard metrics for imbalanced classification, treating the minority *Buggy* class as the positive. We report *precision*, *recall*, and *F1*, along with their macro average (*macro-F1*) for both classes. Because accuracy is misleading when one class dominates, we use two imbalance-aware measures. The *Matthews correlation coefficient (MCC)* ranges from +1 for perfect agreement through 0 for random guessing to -1 for total disagreement [152], and is regarded as the most informative metric for imbalanced binary classification [45]. The *Area Under the Precision-Recall*

Curve (PR-AUC) similarly centers on the minority class, summarizing the precision attainable across recall levels without rewarding correct majority-class predictions. We also report the area under the ROC curve (ROC-AUC) for completeness.

Statistical Tests

To determine whether a difference between two configurations is real rather than sampling noise, we compare distributions of metric values rather than single aggregate scores. We assess significance using the Wilcoxon rank-sum test (also known as the Mann–Whitney U test [154]) at $p \leq 0.05$, and quantify the magnitude of any significant difference with Cliff’s δ [147], a non-parametric effect size, binning it as negligible ($|\delta| < 0.147$), small ($0.147 \leq |\delta| < 0.33$), medium ($0.33 \leq |\delta| < 0.474$), or large ($|\delta| \geq 0.474$) following Hess and Kromrey [105]. We report significance and effect size together, since a significant difference can still practically be negligible in magnitude. Because we compare the two variants across ten metrics at once, we control the false-discovery rate using the Benjamini–Hochberg procedure [25].

When the rank-sum test is inconclusive, we additionally examine the direction of per-project differences using a two-sided sign test [56]. This asks only whether one configuration wins on more projects than the other and detects a consistent advantage that is too small for the rank-sum test to flag. Finally, to compare two models on the same held-out test set rather than two metric distributions, we use McNemar’s test [155] on their paired predictions, counting the instances where the two models disagree and testing whether that disagreement is one-sided. We use this test specifically to check whether the fusion model and the metrics-only baseline differ

systematically rather than by chance.

Evaluation Setup

We implement the traditional ML-based classifiers with scikit-learn [184] and XGBoost [42], the resampling strategies with imbalanced-learn [134], and the fusion network in PyTorch. All experiments were conducted on a High-Performance Computing (HPC) cluster equipped with Intel 8570 CPUs (2.1 GHz) and multiple NVIDIA H100 GPUs. The number of CPUs and GPUs and the memory allocation varied by model, depending on resource availability and scheduling within the cluster.

3.4 Results

In this section we present and analyze our findings based on the two research questions. For both research questions, we present the results progressively, first we present the results for noisy dataset and then less-noisy with tabular features only, and finally the fusion models with metrics and embeddings.

3.4.1 RQ1: Predictability at Inception

Detecting a method’s future bug-proneness at its creation, before any change history is available, can help developers prioritize their efforts. Using the *Inception* dataset family, we evaluate this scenario in three stages: a tabular baseline on the *Noisy* variant, the same baseline on the LLM-refined *Less-Noisy* variant, and a fusion model with source-code embeddings. We report MCC and PR-AUC, which remain

reliable under the severe class imbalance (2.56% *Buggy* in the test set), treating *Buggy* as the positive class.

Baseline on the Noisy dataset

We first train the tabular classifiers with the four resampling strategies on the *Inception Noisy* variant, the data before untangling. Table 3.3 reports the top configurations on the validation set. The complete result with all the metrics can be found in our replication package ([Results/Tables/rq1_1_results.pdf](#)).

Table 3.3: Top performing configurations on the validation set for the *Inception Noisy* dataset.

Classifier	Sampler	MCC	AUC_PR	F1_Macro	F1_Buggy	F1_NotBuggy
MLP	RUS	0.18	0.15	0.51	0.18	0.83
RF	RUS	0.15	0.12	0.49	0.16	0.81
XGB	None	0.15	0.14	0.51	0.17	0.84
XGB	ROS	0.14	0.13	0.50	0.16	0.84
LR	None	0.14	0.11	0.49	0.16	0.82
LR	RUS	0.14	0.11	0.49	0.16	0.82
LR	SMOTE	0.14	0.11	0.49	0.16	0.82
XGB	SMOTE	0.12	0.10	0.56	0.15	0.96
MLP	None	0.09	0.16	0.51	0.05	0.98
RF	SMOTE	0.08	0.11	0.52	0.08	0.97

We take the strongest configuration, an MLP with random under-sampling (RUS),

as the baseline which reaches $\text{MCC} = 0.18$ and $\text{PR-AUC} = 0.15$ on validation set. On the test set it falls to $\text{MCC} = 0.13$ and $\text{PR-AUC} = 0.08$. Undersampling achieves high *Buggy* recall (0.71) at the cost of very low precision (0.05), so almost all flagged methods are false positives and the choice of sampler shifts the metrics only negligibly. Further, a randomized hyperparameter search leaves MCC unchanged and lifts PR-AUC only marginally ($0.08 \rightarrow 0.10$).

These values are modest in absolute terms, but they exceed the performance of a dummy classifier. At 2.56% prevalence, an all-*NotBuggy* classifier scores $\text{MCC} \approx 0$ and $\text{PR-AUC} \approx 0.026$. The baseline clears both by a wide margin.

Effect of untangling - the Less-Noisy dataset

We repeat the identical grid on the *Inception Less-Noisy* variant. Table 3.4 reports the top configurations and the full result table is in our replication package ([Results/Tables/rq1_2_results.pdf](#)).

No single configuration performs best across all metrics, and the performance differences are modest. To maintain a direct comparison with the *Noisy* baseline, we again select MLP+RUS. On the test set, it achieves an MCC of 0.14 and a PR-AUC of 0.11, improving to 0.16 and 0.15, respectively, after randomized hyperparameter search. Both metrics are substantially higher than those on the *Noisy* dataset.

To assess whether label untangling benefits all models rather than a single configuration, we compare all 20 classifier-sampler combinations between the two variants using a Wilcoxon rank-sum test with Benjamini-Hochberg correction. The *Less-Noisy* variant significantly outperforms the *Noisy* variant on six of ten metrics. Cliff's δ

Table 3.4: Top performing configurations on the validation set for the *Inception Less-Noisy* dataset.

Classifier	Sampler	MCC	AUC_PR	F1_Macro	F1_Buggy	F1_NotBuggy
MLP	RUS	0.21	0.19	0.53	0.21	0.85
RF	RUS	0.20	0.16	0.52	0.20	0.84
XGB	ROS	0.19	0.19	0.54	0.20	0.87
XGB	RUS	0.19	0.19	0.51	0.19	0.83
XGB	SMOTE	0.19	0.17	0.59	0.22	0.97
XGB	None	0.19	0.20	0.54	0.20	0.87
LR	RUS	0.18	0.15	0.51	0.18	0.84
LR	SMOTE	0.18	0.15	0.51	0.18	0.84
RF	SMOTE	0.13	0.15	0.55	0.12	0.97
MLP	None	0.12	0.21	0.52	0.06	0.98

indicates large gains in PR-AUC, ROC-AUC, F1-macro, and Precision-Buggy, and medium gains in MCC and F1-Buggy, while the remaining metrics (all *NotBuggy* related metrics and Buggy recall) show negligible to small effects. Notably, every paired comparison favors the *Less-Noisy* variant, indicating that cleaning *Buggy* labels consistently improves minority-class and overall performance without affecting majority-class performance.

Adding code semantics - the fusion model

Finally, we add the source-code embedding as a second modality and train the two-tower fusion classifiers on the Less-Noisy variant. Table 3.5 reports the top configurations and the full table is in our replication package ([Results/Tables/rq1_3.results.pdf](#)).

Table 3.5: Top performing configurations on the validation set of *Inception Less-Noisy* dataset using Fusion models.

Fusion	Sampler	MCC	AUC_PR	F1_Macro	F1_Buggy	F1_NotBuggy
Concat	None	0.26	0.21	0.63	0.29	0.96
Gated	None	0.24	0.19	0.62	0.27	0.96
Attention	None	0.23	0.20	0.61	0.26	0.97
Concat	RUS	0.21	0.20	0.55	0.21	0.88
Attention_Gated	RUS	0.21	0.21	0.53	0.21	0.86
Attention	RUS	0.21	0.20	0.54	0.21	0.86
Film	ROS	0.20	0.17	0.59	0.24	0.94
Film	None	0.20	0.22	0.58	0.19	0.97
Attention_Gated	None	0.20	0.20	0.59	0.20	0.97
Gated	ROS	0.19	0.17	0.59	0.23	0.96

A plain concatenation fusion is best on validation, reaching $\text{MCC} = 0.26$, $\text{PR-AUC} = 0.21$, $\text{F1-macro} = 0.63$, and $\text{F1-Buggy} = 0.29$. A McNemar test on the test-set predictions confirms that this fusion model and the tuned metrics-only *Less-Noisy* baseline disagree systematically, with the disagreement strongly favoring fusion

model (49,353 vs. 4,182 discordant predictions, $p \approx 0$).

On the test set, the result drops again, with 0.18 MCC, 0.12 PR-AUC, and 0.18 F1-Buggy. However, the fusion model retains the highest MCC of the three cases (0.18, vs. 0.16 in *Less-Noisy* baseline and 0.13 in *Noisy* baseline). and the highest F1-Buggy (0.18, vs. 0.12 and 0.10), but its PR-AUC (0.12) and ROC-AUC (0.76) fall below the baseline’s (0.15 and 0.80). Figure 3.3 summarizes the three cases across metrics.

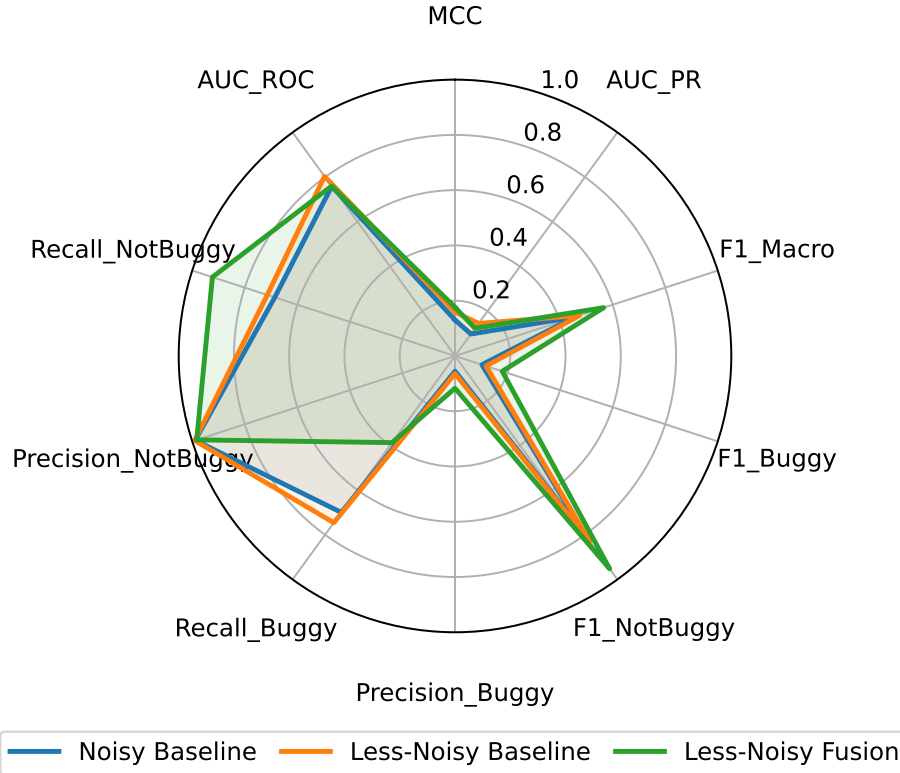


Figure 3.3: Test-set metrics of the baselines and best fusion model on the *Inception* dataset. Results improve gradually, and fusion leads on most metrics.

Why the validation-to-test drop? Across all experiments, performance consistently drops on the test set. Since the test set contains twice as many projects as

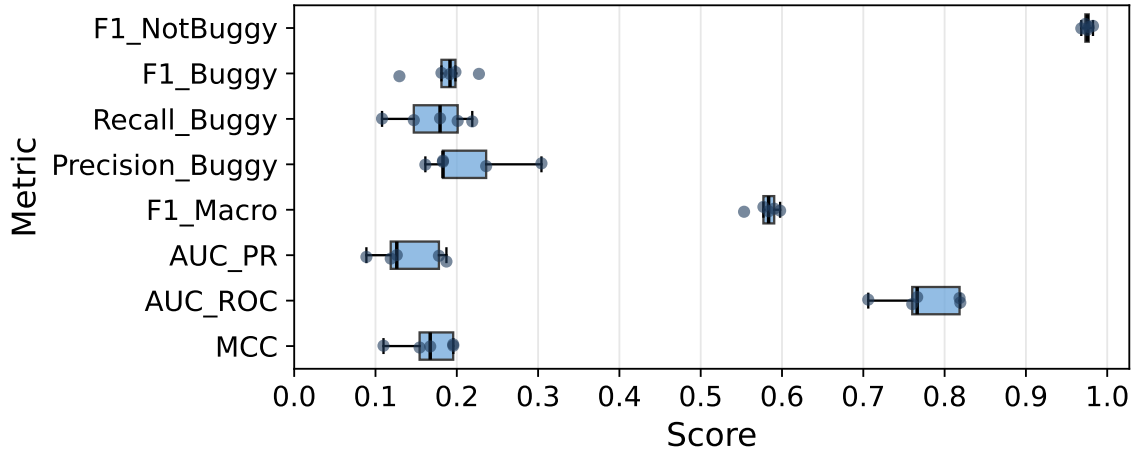


Figure 3.4: Five-fold cross-validation of the best fusion model on the full *Inception Less-Noisy* dataset (≈ 20 projects per fold). The wide fold-to-fold spread shows that project composition, not model quality, drives much of the variation in reported performance.

the validation set (20 vs. 10) and has a lower *Buggy* prevalence (2.56% vs. 4.73%), we hypothesized that this gap is caused by differences in project composition rather than model quality. To investigate, we performed 5-fold project-level cross-validation over the full dataset, with each fold holding approximately 20 projects (Figure 3.4). Each project appears in only one split per fold, preventing data leakage. The results show substantial variation across folds, with MCC ranging from 0.11 to 0.20 and similar variability in the other metrics, indicating that the choice of test projects has a strong influence on the reported performance.

Per-project behavior. We then evaluate the best fusion model separately on each of the 20 test projects (Figure 3.5). The result is encouraging in one respect and sobering in another.

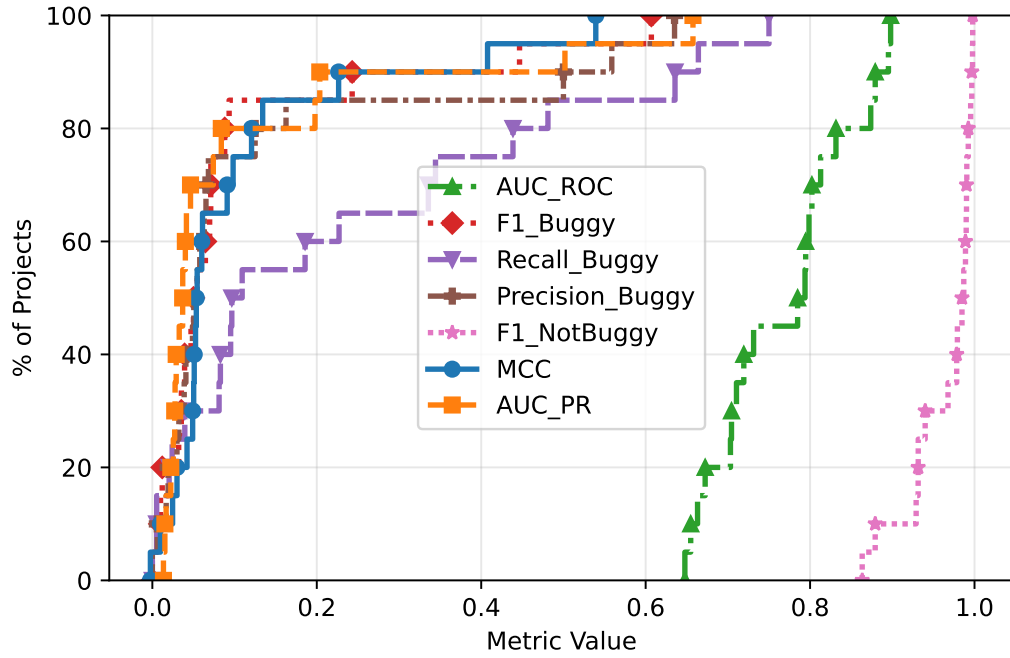


Figure 3.5: CDF of per-project metrics for the best fusion model on the 20 test projects. All projects outperform trivial baselines ($MCC > 0$, $ROC-AUC > 0.5$, $PR-AUC$ above the *Buggy* rate), but performance varies widely. For readability the number of markers is fewer than the number of data points.

Every project outperforms its trivial baseline. All 20 projects achieve $MCC > 0$ and $ROC-AUC > 0.5$, and each project’s $PR-AUC$ exceeds its *Buggy* prevalence. The median per-project MCC is 0.17, close to the pooled value of 0.18. Performance varies substantially across projects. MCC ranges from 0.57 for *nacos*, which also has the highest *Buggy* prevalence ($\approx 19\%$), to 0.08–0.12 for the most imbalanced projects (*mybatis-plus*, *hadoop*, *guava*, and *spring-framework*, each with only ≈ 0.5 – 1.2% *Buggy* methods). These low-performing projects include both the smallest project (*mybatis-plus*, 771 methods) and the largest (*hadoop*, 60.7k methods), suggesting that class

imbalance, rather than project size, is the primary factor. However, the relationship is not strictly monotonic. For example, *xerces2-j* has a relatively high *Buggy* prevalence (15.5%) but achieves an MCC of only 0.14.

RQ1 Summary: Method-level bug prediction at inception is extremely challenging. Untangling lifts *Buggy*-class and aggregate metrics significantly and fused code embeddings raise MCC and F1-Buggy further, though without improving PR-AUC and ROC-AUC. Overall, severe class imbalance and variation across projects remain the primary limitations, leaving inception-time bug prediction far from solved.

3.4.2 RQ2: Predictability Through Evolution

Methods evolve over time. A method’s body and sometimes its entire logic, can change many times after inception. Also a method can become clean or stay buggy even after a bug fix. Detecting bug-proneness at each stage of a method’s evolution is therefore the more operational question. Using the *Evolution* dataset family, we evaluate this scenario in three stages (a tabular baseline on the *Noisy* variant, the same baseline on the LLM-untangled *Less-Noisy* variant, and a fusion model adding code embeddings) and again foreground MCC and PR-AUC, taking *Buggy* as the positive class. The imbalance here is even more severe than at inception: the test set is only 0.94% *Buggy*.

Baseline on the Noisy dataset

We train the tabular classifiers with the four resampling strategies on the *Evolution Noisy* variant, as in RQ1. Table 3.6 reports the top configurations and the full result

table is in our replication package ([Results/Tables/rq2_1_results.pdf](#)).

Table 3.6: Top performing configurations on the validation set for the *Evolution Noisy* dataset.

Classifier	Sampler	MCC	AUC_PR	F1_Macro	F1_Buggy	F1_NotBuggy
MLP	None	0.18	0.12	0.57	0.15	0.99
XGB	SMOTE	0.15	0.06	0.57	0.16	0.98
RF	ROS	0.15	0.11	0.54	0.09	0.99
RF	None	0.14	0.11	0.52	0.06	0.99
DT	ROS	0.12	0.03	0.55	0.13	0.97
XGB	None	0.11	0.12	0.49	0.08	0.90
XGB	ROS	0.11	0.07	0.49	0.08	0.89
MLP	RUS	0.11	0.07	0.47	0.08	0.86
RF	RUS	0.11	0.07	0.47	0.08	0.85
RF	SMOTE	0.10	0.08	0.54	0.09	0.99

The strongest is an MLP with no resampling, reaching $MCC = 0.18$ and $PR-AUC = 0.12$ on validation and falling to $MCC = 0.12$ and $PR-AUC = 0.05$ on the test set. A randomized hyperparameter search helps here, lifting the test scores to $MCC = 0.14$ and $PR-AUC = 0.09$. These numbers are weak, but again non-trivial. At 0.94% prevalence, an all-*NotBuggy* classifier scores $MCC \approx 0$ and $PR-AUC \approx 0.009$; the tuned baseline clears both. Version-level tabular features thus carry a real but faint bug signal, as the inception-time features did.

Effect of untangling: the Less-Noisy dataset

We repeat the same experiments on the *Evolution Less-Noisy* dataset. Table 3.7 presents the top 10 configurations and the full result table is in our replication package ([Results/Tables/rq2_2_results.pdf](#)).

Table 3.7: Top performing configurations on the validation set for the *Evolution Less-Noisy* dataset.

Classifier	Sampler	MCC	AUC_PR	F1_Macro	F1_Buggy	F1_NotBuggy
MLP	None	0.23	0.15	0.58	0.17	0.99
RF	ROS	0.17	0.14	0.55	0.10	0.99
RF	None	0.16	0.13	0.53	0.07	0.99
XGB	SMOTE	0.15	0.08	0.57	0.16	0.98
XGB	None	0.14	0.13	0.50	0.10	0.90
XGB	ROS	0.14	0.08	0.50	0.10	0.90
XGB	RUS	0.14	0.16	0.48	0.09	0.87
RF	RUS	0.14	0.09	0.48	0.09	0.87
RF	SMOTE	0.14	0.11	0.55	0.12	0.99
MLP	RUS	0.13	0.09	0.47	0.08	0.85

The MLP without resampling leads on most metrics, reaching $MCC = 0.23$ and $PR-AUC = 0.15$ on validation (+0.05 and +0.03 over the Noisy baseline) before dropping to $MCC = 0.15$ and $PR-AUC = 0.09$ on the test set. Here, in contrast to the Noisy variant, tuning does not improve the result.

To test whether untangling helps in general, we compare the two variants' configurations (5 classifiers $\times$ 4 samplers) per metric with a Wilcoxon rank-sum test under Benjamini–Hochberg correction, as in RQ1. The result is weaker than at inception. Only ROC-AUC clears significance, and it does so with a large effect. The direction, however, is consistent. The Less-Noisy variant is higher in 17 to 18 of the 20 paired comparisons for these metrics, and the sign test is significant for them. Untangling therefore provides a consistent, if modest, benefit for version-level bug prediction.

Adding code semantics: the fusion model

Finally, we add the code embedding and train the two-tower fusion classifiers on the Less-Noisy variant. Table 3.8 reports the top configurations and the full result table is in our replication package ([Results/Tables/rq2.3.results.pdf](#)).

A gated fusion is best on validation, leading on most metrics with $\text{MCC} = 0.24$ and $\text{PR-AUC} = 0.17$. On the test set, however, the result falls again dramatically to $\text{MCC} = 0.12$ and $\text{PR-AUC} = 0.05$, below the tuned tabular baselines. Figure 3.6 summarizes the three cases across metrics.

Why the validation-to-test drop? To investigate the large drop in test performance, we perform the same analysis as in RQ1. The results show that test performance again depends strongly on the projects included in the test split (Figure 3.7).

The 5-fold project-level cross-validation shows that across folds, the MCC is low and variable (median ≈ 0.09 , range 0.06 to 0.16) and other metrics also show the same trend. The fold-to-fold variance is large relative to the mean, confirming that project composition drives much of the headline number.

Table 3.8: Top performing configurations on the validation set of *Evolution Less-Noisy* dataset using Fusion models.

Fusion	Sampler	MCC	AUC_PR	F1_Macro	F1_Buggy	F1_NotBuggy
Gated	None	0.24	0.17	0.62	0.24	0.99
Concat	None	0.20	0.15	0.60	0.21	0.99
Attention_Gated	None	0.19	0.15	0.59	0.20	0.99
Attention	None	0.18	0.14	0.58	0.18	0.99
Attention_Gated	ROS	0.18	0.15	0.57	0.17	0.97
Film	None	0.17	0.12	0.58	0.17	0.99
Film	ROS	0.14	0.08	0.57	0.15	0.98
Attention	RUS	0.14	0.14	0.52	0.11	0.93
Concat	ROS	0.13	0.08	0.56	0.15	0.98
Gated	ROS	0.12	0.08	0.56	0.13	0.99

Per-project behavior. Evaluating the best fusion model separately on each of the 20 test projects (Figure 3.8) reveals the same pattern as RQ1, although with lower performance. Nearly every project outperforms its trivial baseline: 19 of 20 projects achieve $MCC > 0$ (the exception is *dropwizard*, with $MCC = -0.002$), all achieve $ROC-AUC > 0.5$, and every project’s PR-AUC exceeds its *Buggy* prevalence. Despite this, project-level performance remains low and varies widely. The median MCC is only 0.05, ranging from 0.54 for *nacos* and 0.41 for *xerces2-j*, both with approximately 14% *Buggy* methods, to nearly zero for projects with the lowest *Buggy* prevalence. As in RQ1, these results suggest that prediction performance is driven

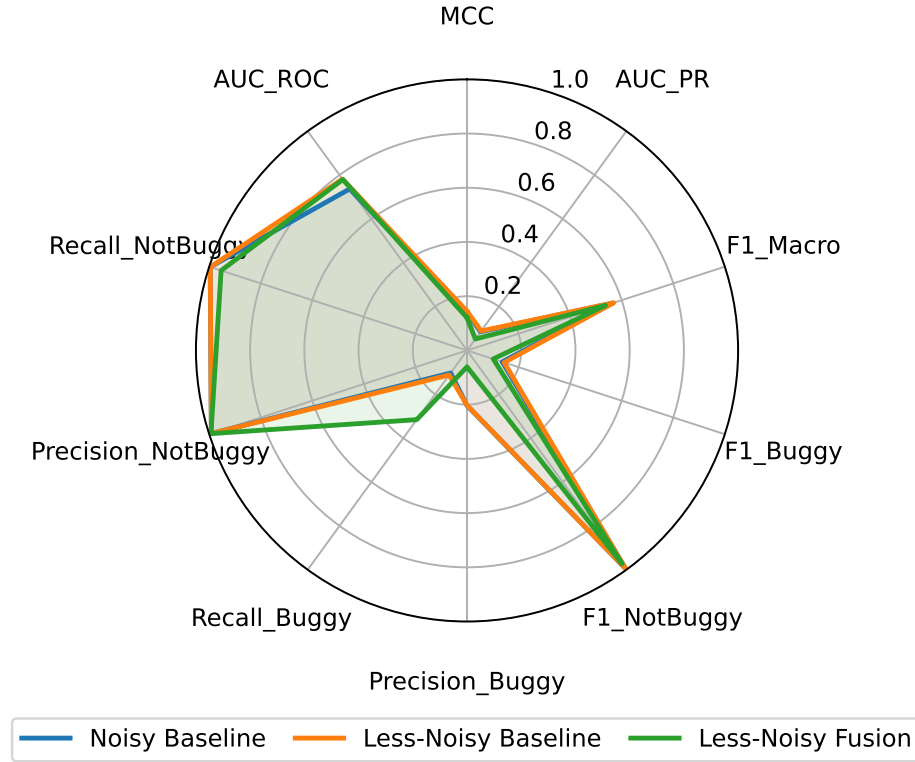


Figure 3.6: Test-set metrics for baselines and the best fusion model on the Evolution family. Fusion and the tabular less-noisy baseline have similar performance, with fusion providing higher buggy recall.

by class imbalance.

Version-wise temporal split. All the results use the project-disjoint split, which imposes a cross-project cold start. The Evolution family admits the “train-on-past, predict-future” protocol using the version-wise temporal split. Table 3.9 shows the performance at different split fractions.

Using the same gated fusion model, the temporal split achieves MCC values of 0.20, 0.224, and 0.217 with 30%, 50%, and 70% training data, respectively, while PR-AUC reaches 0.182. Performance changes little as the training fraction increases,

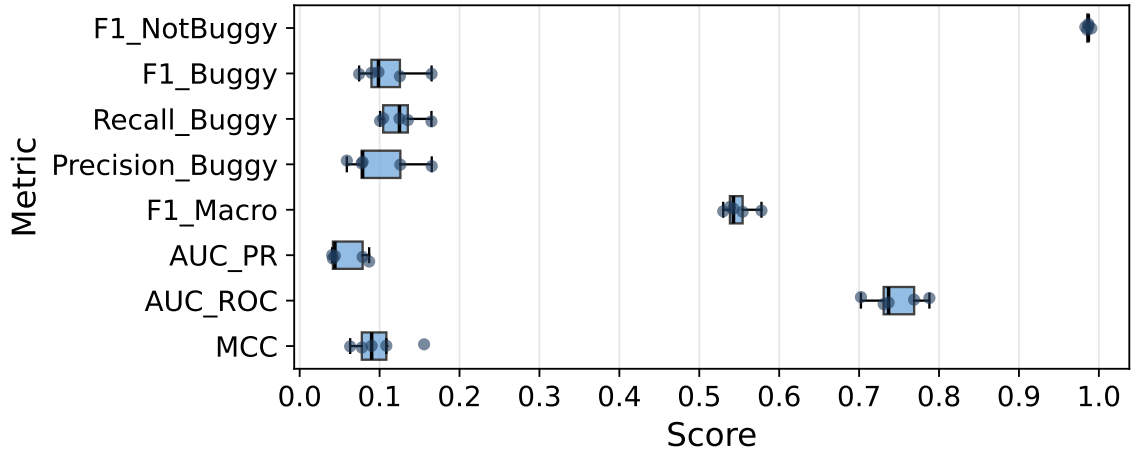


Figure 3.7: Five-fold cross-validation of the best fusion model over the full Evolution Less-Noisy dataset (≈ 20 projects per fold). The wide fold-to-fold spread shows that project composition, not model quality, drives much of the variation in reported performance.

Table 3.9: Scores on test set for different type of temporal split on the Evolution Less-Noisy dataset. 56.41% of buggy samples from the entire dataset is in early 30% of the data.

Train%	Buggy%	MCC	AUC_PR	F1_Macro	F1_Buggy	F1_NotBuggy
50	67.31	0.224	0.182	0.612	0.232	0.991
70	75.36	0.217	0.181	0.608	0.230	0.987
30	56.41	0.200	0.143	0.598	0.204	0.992

indicating that the available learning signal saturates early because more than half of the *Buggy* samples appear within the first 30% of the data. These results are substantially better than those obtained with the project-disjoint split (MCC = 0.12,

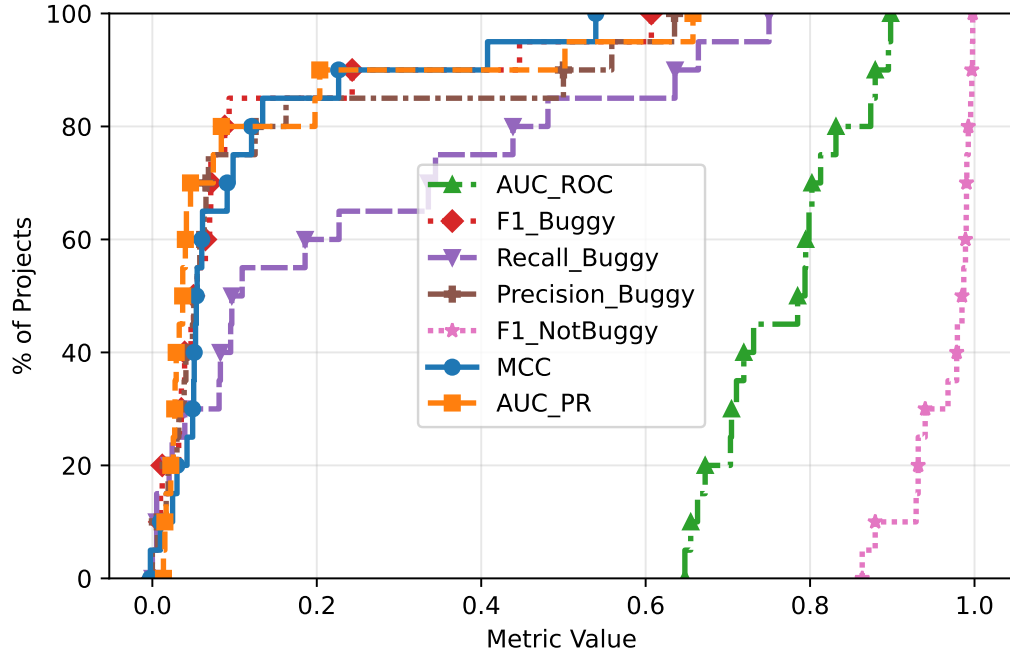


Figure 3.8: CDF of per-project metrics for the best fusion model on the 20 Evolution test projects. Nineteen of twenty clear $MCC > 0$, but the median per-project MCC is only 0.05. For readability the number of markers is fewer than the number of data points.

PR-AUC = 0.05). However, the two evaluation protocols are not directly comparable. The temporal split allows earlier versions of the same method to inform predictions for later versions, whereas the project-disjoint split requires generalization to entirely unseen projects. The large performance gap therefore suggests that differences between projects are the primary challenge to model generalization.

RQ2 Summary: Version-level bug-proneness prediction is as challenging as inception-level prediction. Untangling and code embeddings improve performance on validation set, but the gains do not generalize across projects. Performance remains limited by class imbalance, while the stronger temporal-split results suggest that project-specific models are more effective than cross-project models. Overall, version-level prediction remains far from deployment-ready.

3.5 Discussion

Across both research questions, the results converge on a single story: method-level bug prediction remains challenging, even after addressing the field’s known obstacles. Untangling improves prediction in both dataset families, and fused code embeddings add further lift. Yet neither generalizes across projects, and per-project accuracy varies widely, reflecting each project’s bug rate. In this section, we draw four implications from our findings, and based on them, we discuss possible future works.

3.5.1 Implications

Cleaning labels is worth the effort

Removing tangled noise with an LLM improved Buggy-class and aggregate metrics across both dataset families. We can be fairly confident that this gain comes from label quality rather than a shift in class balance, because the Noisy and Less-Noisy variants are matched in class counts. This suggests the LLM-based untangling of Chapter 2 is a practical way to clean datasets at scale and cleaner labels yield measurably better

predictors.

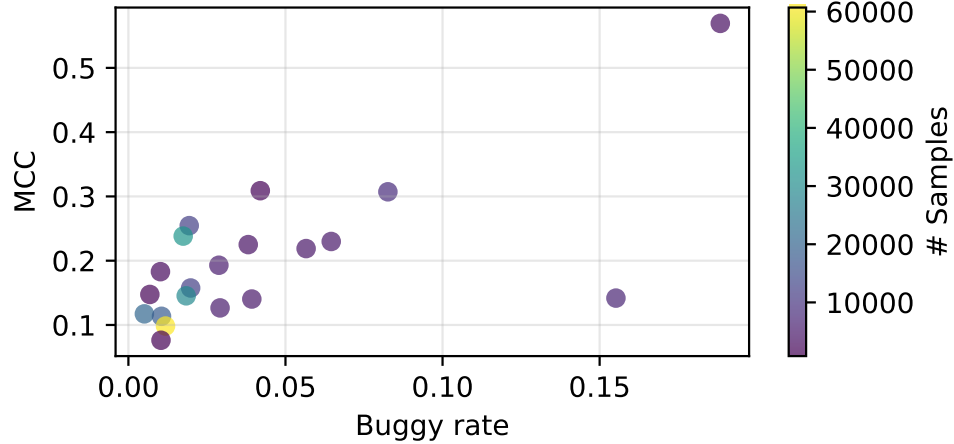


Figure 3.9: Per-project MCC and buggy rate for the fusion model on the *Inception* dataset. Buggy rate, not sample size, drives MCC.

Class imbalance, not project size, is the dominant limit

Per-project prediction performance is driven more by the prevalence of buggy methods than by project size (Figure 3.9). Models consistently perform better on projects with a higher proportion of buggy methods, while performance degrades on highly imbalanced projects regardless of the number of methods they contain. This pattern holds across both the inception and evolution settings, suggesting that extreme class imbalance—not project scale—is the primary obstacle to accurate prediction. Our results further indicate that incorporating imbalance handling directly into the learning algorithm can be as effective as, or more effective than, resampling techniques, highlighting imbalance-aware learning as a more promising direction.

Project composition drives the headline number

Five-fold cross-validation over the full datasets exposes a wide fold-to-fold spread, and our validation and test splits differ in Buggy prevalence (4.73% vs. 2.56% at inception). Much of the variation in any single reported score is therefore an artifact of which projects land in the test split. In other words, one train-test split is not enough to judge a model; researchers should report results across several folds and for individual projects rather than relying on a single number.

Within-project signal beats cross-project

Model performance is substantially higher under the temporal split than under the project-disjoint split, highlighting the difficulty of cross-project generalization. Unlike the project-disjoint setting, the temporal setting benefits from historical information about the same methods, allowing models to exploit project-specific patterns that are unavailable in a cold-start scenario. This observation is consistent with prior work showing that time-aware evaluation is more realistic and generally more favorable than cross-project prediction [19, 51].

3.5.2 Future Work

Our results show that, even after reducing label noise and incorporating multi-modal features, method-level bug prediction remains a challenging problem. The two primary limitations are severe class imbalance and limited cross-project generalization, suggesting that further improvements are unlikely to come from cleaner labels or additional features alone.

One promising direction is imbalance-aware learning, including cost-sensitive learning and bug augmentation using generative models to synthesize realistic buggy methods. These approaches could improve the representation of the minority class and enhance model robustness. Another direction is project-specific adaptation, where a model is pre-trained on many projects and then fine-tuned using the history of the target project to better capture project-specific characteristics. Finally, Large Language Models (LLMs) offer a promising alternative to conventional classifiers. By reasoning over both source code and its historical evolution, LLMs may better capture semantic and temporal information, leading to more accurate method-level bug prediction.

3.5.3 Threats to Validity

External validity. Our study considers only Java open-source GitHub projects. Consequently, the findings may not generalize to closed-source software or projects written in other programming languages.

Construct validity. Method history recovery can be inaccurate under substantial refactoring. We identify bug-fixing commits using a keyword-based heuristic that may seldom produce wrong results. In addition, methods are labeled *NotBuggy* if no bug-fixing commit is observed within five years, extending the two-year window used by [51]. However, they may still contain latent defects that are fixed beyond our observation window.

Internal validity. The construction of the Less-Noisy dataset relies on the open-weight LLM *gpt-oss-20b* to untangle bug-fixing commits. Errors in the untangling

process may incorrectly retain or remove *Buggy* labels. It can also be hampered by our choice of the statistical tests: the Wilcoxon rank sum and Cliff’s delta. These two tests, however, are widely adopted in software engineering research [19, 48, 49, 60].

Conclusion validity. The conclusion validity of our findings can be impacted by any of the above-mentioned threats.

3.6 Conclusion

We presented a large-scale method-level bug-prediction study that confronts the field’s three principal obstacles—inaccurate labeling, unrealistic evaluation, and shallow features—within a single pipeline over 100 open-source Java projects. We addressed label quality by labeling methods conservatively and removing tangled noise with an LLM, producing a Noisy and a Less-Noisy variant of every dataset; we addressed evaluation realism by studying two prediction settings, Inception and Evolution, the latter under a temporal split; and we addressed feature shallowness by fusing Qwen3 source-code embeddings with tabular metrics in a two-tower network. While untangling and code embeddings yielded consistent improvements in both prediction settings, the best-performing models reached only ≈ 0.15 – 0.18 MCC on the test data. Cross-project generalization also remained limited, and per-project MCC closely followed the prevalence of buggy methods. Establishing this ceiling under a rigorous, realistic, and embedding-augmented pipeline is itself our contribution: it signals that the remaining gap is unlikely to be closed by cleaner labels or richer features alone, pointing instead toward imbalance-aware learning and per-project adaptation.

Part II

App Popularity Prediction

Part II moves from an internal outcome to an external one. Bugs live inside the code, but popularity is decided by users and markets, far beyond the reach of source code alone. Chapter 4 asks whether the same kinds of internal metrics, measured at an app's launch, still carry a useful signal for its ratings and downloads, and finds that the answer depends on how the question is framed.

Chapter 4

Moderately Mighty: To What Extent Can Internal Software Metrics Predict App Popularity at Launch?

Abstract

Predicting a mobile app's popularity before its first release can give developers a strategic advantage, yet it remains challenging. This study explores the extent to which internal software metrics, measurable from source code before deployment, can predict an app's popularity (i.e., user ratings and downloads per year) at its inception. We built a rigorously filtered dataset of 446 open-source, Java-based Android apps available on both F-Droid and the Google Play Store. Using source code from F-

Droid, we extracted system-, class-, and method-level code metrics, code smells, and app metadata. Popularity information, including reviews and download counts, was collected from Google Play.

We evaluate regression and classification models using three feature sets: a minimal **Size-only** baseline, a domain-informed **Handpicked** set, and a **Voting** set produced by combining several feature-selection algorithms. Across both targets (ratings and downloads), regression models perform poorly, driven by a skewed rating distribution and a highly scattered range of download counts. When reframed as a binary classification task (Popular vs. Unpopular), performance improves significantly—the best model, a Multi-Layer Perceptron, achieves an F1-score of 0.72.

We conclude that, although internal code metrics alone are insufficient for accurately predicting an app’s future popularity, they do exhibit meaningful correlations with it. Thus, our findings challenge prior studies that have entirely dismissed internal metrics as valid indicators of software quality. Instead, our results align with research suggesting that internal code metrics can be valuable when evaluated within the appropriate context—specifically, we found them useful for classification tasks.

4.1 Introduction

The rapid evolution of handheld devices, such as smartphones and tablets, along with the development of mobile operating systems, has broadened the scope, complexity, and opportunity in mobile application development. As the user base continues to grow, so does the demand for mobile applications, leading to an increase in the number of vendors [76]. Mobile application developers primarily rely on prominent

platforms such as Google Play and Apple’s App Store to distribute their applications to end users. These platforms provide a centralized marketplace where developers can sell or distribute their products, and users can compare applications, read reviews and ratings, and make informed decisions—essentially serving as “one-stop shops” [256]. Users contribute to the marketplace by providing reviews and ratings that reflect their level of satisfaction or dissatisfaction with an application based on personal experience.

When choosing applications, users consider a variety of factors. While features such as being *ad-free* or *free-to-use* are appealing, the most influential indicators of an app’s popularity are its user reviews (e.g., ratings) and the number of downloads [8, 141]. These metrics play a critical role not only in shaping user perception but also in determining the commercial success of developers who rely on app sales for income.

Given the strong connection between a developer’s success and an app’s popularity, we argue that developers would greatly benefit from the ability to predict an app’s popularity prior to its release. Such foresight is especially valuable before launching the initial version, as users often form lasting impressions based on their first experience [66, 107, 159]. Additionally, the lower number of early downloads and user ratings can significantly influence app store ranking algorithms [120], affecting the app’s visibility and making it more difficult to attract users later—even if the app is improved. Unfortunately, existing relevant work required history data while analyzing and predicting app popularity, making such analysis impractical for popularity prediction at an app’s inception [38].

Previous research [9] observed that Android users participate more in rating apps

than iOS users, making the Android app stores more suitable for our study. Leveraging data from these stores, this study investigates the extent to which app popularity can be predicted using only internal software metrics available at the early stages of development—specifically, at an app’s inception. Prior research has demonstrated that internal metrics can reliably estimate external software quality attributes, such as bug-proneness, change-proneness, and the ease of extending functionality. Given that these external factors are strongly linked to app popularity, we pose the following question: *can internal software metrics be used to predict key indicators of popularity, such as user ratings and download counts?*

While app ratings and download counts are readily available on the Google Play Store, measuring internal software metrics requires access to the app’s source code. Unfortunately, the Play Store does not provide source code, making it infeasible to extract these metrics directly from that platform. To overcome this limitation, we utilized the F-Droid repository¹—a collection of open-source Android applications. By mapping F-Droid apps to their counterparts on the Play Store, we were able to collect both independent (internal software metrics) and dependent variables (ratings and downloads), enabling us to build predictive machine learning models. For internal metrics, we focused on popular code metrics at three granularity levels: system level (e.g., DL score), class level (e.g., C&K metrics), and method level (e.g., McCabe). Additionally, we extracted commonly used code smells (e.g., excessive dependency) to capture aspects of code and design quality. Since app popularity may also be influenced by factors such as feature count, app genre, and the presence of advertisements, we incorporated these attributes into our analysis as well.

¹<https://f-droid.org/en/> last accessed: 14 April 2025

We aim to investigate the potential of internal code metrics in predicting app popularity, using two widely adopted indicators: user rating and download count. Accordingly, our study is guided by the following two research questions.

RQ1: *To what extent can internal software metrics predict an app's user rating?*

To address this question, we first applied different regression algorithms to predict an app's average user rating based on its internal software metrics. Our results indicate that internal software metrics measured from the initial app version perform very poorly in predicting an app's future rating. However, to observe if internal metrics have *any* impact on app popularity, we reformulated the problem as a binary classification task—labeling apps as *Popular* or *Unpopular* based on a predefined rating threshold. In this setting, the results are more promising with an F1-score of 0.72. This suggests that an app's internal code and design quality are important factors in eventual app popularity, but not the only important factor. This is not surprising given that app popularity may also depend on aspects such as the types of features offered and how pleasing its UI is, which are unlikely to be correlated with any internal software metrics.

RQ2: *To what extent can internal software metrics predict an app's number of downloads?*

Following a similar approach to RQ1, we first conducted regression analysis on the yearly download count (i.e., downloads per year) rather than the total number of downloads, as the latter can be heavily influenced by an app's age. Consistent with our findings for user ratings, we observed that predicting the exact number of

yearly downloads is extremely challenging. However, when we reframed the task as a binary classification problem—i.e., distinguishing between *Popular* and *Unpopular* apps based on a predefined number of downloads—the results are more encouraging with an F1-score of 0.69.

In summary, while internal metrics alone are insufficient to accurately predict an app’s future popularity, they do offer meaningful predictive insights (hence the title, *moderately mighty*). This finding challenges earlier studies [67, 83] that dismissed the utility of such metrics entirely. In line with another stream of research [50, 130], we find that *context is king* when evaluating internal metrics—observing a negative result in one context (e.g., regression, in our case) does not preclude encouraging results in others (e.g., classification). Our results, therefore, reaffirm the importance—albeit in a limited capacity—of monitoring a variety of code-quality indicators. By detecting early warning signs, developers can proactively improve their apps, potentially saving time, resources, and reputation. Similarly, app stores could leverage internal metrics to enhance ranking algorithms and allocate support more effectively.

To enable replication and extension, we share our data publicly.²

The remainder of this chapter is structured as follows: related work is discussed in Section 4.2; methodology is described in Section 4.3; and approach, analysis, and results for the research questions are presented in Section 4.4. We discuss the implications, threats and future works in Section 4.5. Finally, we conclude this chapter in Section 4.6.

²<https://github.com/SQMLab/AppPopularityPrediction>

4.2 Related Work and Motivation

In this section, we review previous research on App Store analysis. We establish why building internal metrics-based models for predicting app popularity is still an open research problem. We then discuss related work showing the potential of code metrics, code smells, and app metadata in building such models.

4.2.1 App Popularity Analysis

Ali et al. [9] analyzed apps available on both the Play Store and App Store, finding that Android users are more active in rating apps than iOS users. Their study also revealed that majority of the most popular apps are available across both platforms. In a separate analysis of the App Store, Harman et al. [96] reported no significant correlation between an app's price and its rating or popularity. Nayebi et al. [168] highlighted the value of incorporating user feedback from external sources, such as Twitter, in addition to App Store reviews. Ickin et al. [109] observed that while users often rely on reviews when choosing to install apps, they tend to uninstall them due to issues like frequent crashes, intrusive advertisements, and lack of ongoing maintenance, such as feature updates. The problem of crashes and lack of ongoing maintenance issues were also observed by Khalid et al. [122] while analyzing common complaints in app reviews.

Corral et al. [58] examined the influence of C&K metrics on mobile app popularity, finding only a weak correlation. However, their study relied on displayed star ratings, an inaccurate approach as ratings can vary by region, app version, and device type [85, 15]. Similarly, Gezici et al. [80] reported no significant association between internal

code quality and external app quality. In contrast, Cruz et al. [59] argued that thorough testing can enhance code quality, which may in turn improve user ratings. The relationship between code quality and app popularity also appears to be app category-dependent, as observed by Sorbo et al. [64]. Noei et al. [170] found that both device-specific attributes (e.g., memory capacity, display size, battery life) and app-specific characteristics (e.g., UI complexity, code size) can influence user-perceived quality in Android apps. Similarly, Tian et al. [222] observed that, compared to low-rated apps, high-rated apps tend to be larger in size, exhibit greater complexity, and benefit from more extensive marketing efforts. Interestingly, Kuttal et al. [129] highlighted a disconnect between developer and user perspectives: they found a weak correlation between an app's popularity on GitHub and its ratings in the App Store.

Gezici et al. [79] showed that user ratings can be accurately predicted from user reviews using neural sentiment analysis. Rahman et al. [191] found that static code metrics can be employed to build models for predicting privacy and security issues in Android apps. In particular, using a radial-based support vector machine algorithm, they achieved a precision of 0.83. However, this high precision came at the expense of lower recall. Similar results were reported by Alenezi et al. [7], who also used code metrics to predict risk scores.

Grano et al. [87] extracted various code metrics and code smells from decompiled APKs to support research on software evolution and quality improvement. Their methodology involved collecting app versions and user reviews from F-Droid and Google Play, and assessing code smells and quality indicators using automated tools. Catolino [38] built upon the dataset of Grano et al. [87] to examine whether code

quality could be used to predict app ratings. Although the objectives of Catolino’s study align with ours, it is a brief (2-page) work without much in-depth analysis and suffers from two significant methodological issues, impacting the reliability of its findings.

- The study relied on the dataset of Grano et al. [87], which derived code quality metrics from decompiled APKs. However, decompiled code is not equivalent to the original source code, as compilers introduce various optimizations during the APK build process [242, 245]. Consequently, critical code metrics—such as code readability—may differ significantly between the original and decompiled versions. More concerningly, decompiled code often differs from the original source code both syntactically and semantically [97], thereby compromising the validity of any analysis based on metrics derived from it.
- To evaluate model performance, Catolino adopted a 10-fold cross-validation approach. This method is fundamentally flawed in scenarios involving multiple versions of the same entity, which is true for the used dataset. In such scenarios, a 10-fold cross-validation approach introduces data leakage: some versions may end up in the training set while others appear in the test set. This leakage leads to overestimated model performance. Pascarella et al. [182] and Chowdhury et al. [51] have examined this issue in detail. Both studies revisited highly accurate method-level bug prediction models and demonstrated that their performance dropped significantly when evaluated using more realistic approaches that prevent overlap between training and test data.

Based on our previous discussion, we identified a subset of research that explored

correlations between app attributes and app popularity. However, these findings were often inconsistent, with both positive and negative correlations reported. Moreover, only a limited number of studies attempted to develop predictive models based on app attributes, and those efforts were undermined by methodological flaws that compromised their reliability. *Motivated by these gaps, we conduct a new, rigorous study to assess how accurately app popularity can be predicted using only internal software metrics. Importantly, we focus exclusively on the first release of each app, ensuring that our model does not rely on historical data, making it particularly valuable for developers seeking to build a strong reputation from the outset.*

4.2.2 Code Metrics

Code metrics—quantitative measures derived from analyzing source code—have long served as a foundation for assessing software quality and maintainability [153, 12, 182, 51, 151, 74, 10]. A key advantage of code metrics is their availability throughout the software development life cycle, which allows for the construction of predictive maintenance models even in the absence of historical data, effectively addressing the cold start problem [47, 182]. Researchers have developed and studied code metrics at different granularities while analyzing their effectiveness in understanding software maintenance.

System-level metrics. System-level metrics play a pivotal role in evaluating the long-term maintainability of software systems [142]. These metrics provide valuable insights into the overall system architecture and its degradation patterns over time [160, 189, 142]. Architectural issues at the system level not only hinder quality

but also escalate the cost and complexity of future enhancements. To address these challenges, researchers have proposed various system-level metrics. For instance, Sethi et al. [203] introduced the Independence Level metric, while Mo et al. [160] developed the Decoupling Level metric. Both aim to assess how effectively a software system can be modularized into independent components. Similarly, MacCormack et al. [148] proposed the Propagation Cost metric, which measures the degree of coupling among source files within a system. Empirical studies have shown that systems with low decoupling scores tend to be more bug-prone and harder to extend with new functionalities [160]. Further advancing this area, Cai et al. [34] discussed novel architecture-level maintainability metrics and identified architectural design anti-patterns that serve as early warnings of potential structural issues. These anti-patterns assist developers in detecting and resolving architectural flaws before they evolve into critical maintenance challenges.

Class-level and method-level metrics. Some of the most widely adopted class-level metrics are the Chidamber and Kemerer (C&K) metrics [46], which include measures such as depth of inheritance, coupling between objects, and number of children. Basili et al. [21] demonstrated that C&K metrics are effective predictors of class fault-proneness during the early stages of software development. Similarly, Li et al. [136] found these metrics useful for estimating maintenance effort. A broader range of studies supporting the predictive power of C&K metrics for maintainability was summarized by Iftikhar et al. [110]. However, not all findings were consistently positive—some studies, such as Gil and Lalouche [83], reported limited usefulness of class-level metrics in estimating future maintenance efforts.

In recent years, there has been a growing focus on more fine-grained, method-level code metrics [51, 47, 182, 50], largely driven by developers’ interest in insights at that level of granularity. While Pascarella et al. [182] reported that method-level metrics performed poorly in method-level bug prediction, Chowdhury et al. [51, 47] found these metrics to be effective indicators of method-level bug-proneness and change-proneness. Additional studies also supported these findings [130, 81, 161], reinforcing the value of method-level metrics for software quality assessment.

Encouraged by this body of work, we investigate whether code metrics across these three levels of granularity—system, class, and method—can be used to predict app popularity, given their demonstrated success in predicting other software quality and maintainability attributes.

4.2.3 Code Smells

Code smells are sub-optimal design choices in source code that violate established design principles, reducing code quality and maintainability. While their effects may not be immediately visible, over time, they make the codebase harder to understand, modify, and maintain [249]. The software engineering community widely recognizes the negative implications of code smells, which often result in higher maintenance efforts [238, 1]. Given their significant long-term impact, code smell detection has become one of the most extensively studied topics in the field of software quality and maintenance [249]. As such, a stream of research focused completely on developing code smell detection techniques and tools [162, 205, 176]. While studies exist that found minimal or no impact of code smells, the majority of studies agree

that code smells adversely affect software maintenance [123, 217, 247, 178, 177, 124]. For example, Khomh et al. [123] found that classes with code smells tend to be more change-prone than those without any smells. While numerous studies showed the negative impact of source code smells on change- and bug-proneness, Spadini et al. [217] found that code smells negatively impact the change- and bug-proneness of test code as well.

Recognizing the already established quality and maintenance impact of code smells, we include different types of code smells in our study while predicting app popularity.

4.2.4 Other Metadata

In addition to code-level characteristics, prior research has identified several app-level metadata attributes that significantly influence app popularity. These include genre, advertising policy, permissions requested, and the richness of feature sets [222, 90, 230]. These factors reflect design, usability, and monetization trade-offs that are often perceptible to users even before installation, thereby shaping their preferences and satisfaction levels.

App category or genre is a high-level descriptor of an app's intended functionality and target audience. Several studies have examined the distributional effect of genre on user ratings, downloads, and ad strategies. Tian et al. [222] found that the genre or category of an app is the fourth important factor that impacts the app's rating. Similarly, Mahmood et al. [149] observed that genre is one of the most influential variables and has a high impact on the ratings of apps. These findings collectively indicate that app popularity cannot be decoupled from the genre context in which it

operates.

Monetization strategies, particularly the inclusion of in-app advertisements, have also been shown to affect user reception. Gui et al. [90] conducted an empirical study demonstrating that in-app advertisements significantly increase CPU, energy, and network resource consumption, leading to user complaints and lower ratings. He et al. [99] further quantified this effect, showing that apps embedding more than six ad libraries are substantially more likely to receive negative reviews. These findings underscore the tension between revenue generation and user experience, suggesting that an ad-free policy—or a minimally invasive one—can be a distinguishing factor for app popularity.

Permissions requested by an app often raise user privacy concerns, especially when they appear excessive or unrelated to the app’s core functionality. Wang et al. [230] leveraged user reviews to investigate how users rationalize permission requests, finding that unexplained or intrusive permissions negatively affect user trust and ratings. Jisha et al. [115] incorporated permission-based risk assessments into app recommendation systems, arguing that lower-risk apps are generally perceived more favorably.

Feature richness, often inferred from app descriptions or inferred capabilities, has also been explored as a determinant of user appeal. Sarro et al. [198] and Liang et al. [140] showed that the number and diversity of claimed features are highly predictive of user ratings. Using NLP and topic modeling, these studies demonstrate that users are more likely to positively rate apps that advertise functional depth and variety, provided these features align with user needs and are well-integrated.

Motivated by this body of evidence, we systematically captured genre, advertise-

ment policies, requested permissions, and feature richness as part of our metadata collection. These attributes offer complementary explanatory power beyond code-level metrics, enabling a more holistic prediction of app popularity.

4.3 Methodology

In this section, we describe our process for app collection, the methodology used for extracting and selecting source code features, the approach for gathering popularity indicators, and the machine learning algorithms employed to build the prediction models. To maintain clarity and readability, methods specific to individual research questions are presented separately in their respective sections (Section 4.4). Figure 4.1 shows the high-level view of our methodology.

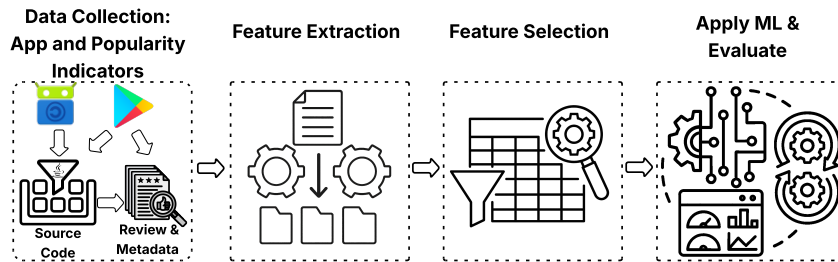


Figure 4.1: Overview of methodology that involves app collection (with source code and ratings), feature extraction, feature selection, and building predictive models.

4.3.1 App Collection

As we focus exclusively on Android apps, we selected the Google Play Store for collecting popularity metrics—specifically, ratings and number of downloads. Unfor-

Unfortunately, the Play Store does not provide access to source code. Inspired by earlier works in this area [87, 245, 57], we leveraged the F-Droid repository³ to collect the source code of the apps. Since our aim is to predict an app’s popularity at its inception, we required the earliest available version of each app. F-Droid maintains two repositories: the main repository, which contains the latest three versions of each app, and an archive that stores the remaining versions. We therefore collected information—including package name, source code, GitHub repository URL, and so on—from the oldest available version in the archive, which represents the first release of an app [87]. Corresponding information, including popularity indicators, genre, advertisement-related information, release date, and other metadata, was retrieved from Google Play. To map apps between F-Droid and Google Play, we used the package name as a common identifier, as the package name, also known as the application ID, is unique for every application⁴. This approach has been commonly used in other studies where mapping apps was necessary [87, 78, 57]. If an app was not available on both platforms, it was excluded from our dataset. After this step, we obtained a collection of 1,347 unique apps.

The apps were developed using various programming languages, such as Java, Kotlin, C#, and C++. Unfortunately, the tools we used to extract software metrics—e.g., class-level C&K metrics—support only Java. Also, it is not recommended to mix multiple programming languages in code metrics-based studies, as code metrics distributions can differ significantly across programming languages [246, 50]. Consequently, we excluded apps developed in non-Java languages. F-Droid provides a

³<https://f-droid.org/en/> last accessed: Apr 18, 2025

⁴<https://developer.android.com/build/configure-app-module>

link to the source code repository for each application, with these repositories hosted across various platforms—predominantly GitHub, but also including GitLab, Codeberg, BriarProject, and others. Each repository typically features a user interface element that displays the programming languages used in the project along with their corresponding percentage distributions. To systematically extract this information, we developed a web crawler that navigates to each repository and parses the language composition data. From the collected dataset, we applied a filtering criterion: only those projects in which Java accounts for at least 50% of the codebase were retained for further analysis. This ensured that our code metric measurements reflected the dominant portion of each app. After this filtering step, 553 apps were retained for analysis.

4.3.2 Popularity Indicators

User-provided star ratings and download counts are key indicators of app popularity [8, 141]. However, the rating displayed on the Play Store is not the actual rating. The displayed rating is not the average of all ratings and also varies by region, device type, and app version [85]. In fact, Ruiz et al. [196] reported that app store ratings do not consistently reflect changes in user satisfaction. As such, we decided to calculate the average rating instead of choosing the displayed rating. To do this, we collected all the user reviews for an app. A user review contains data related to star rating, user’s comment, developer’s reply, corresponding app version, etc.

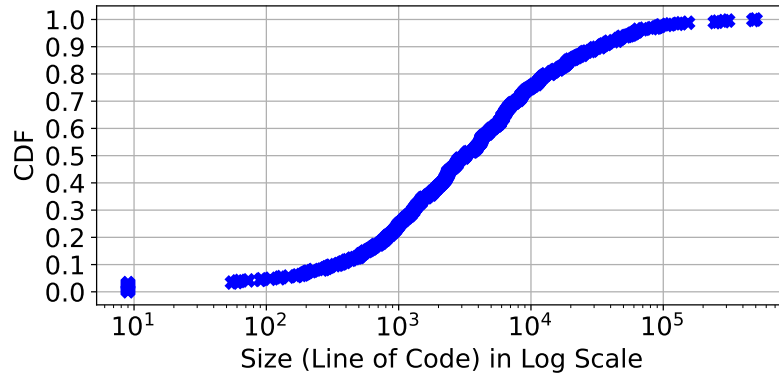
The use of download count as a standalone metric for assessing application popularity is inherently problematic due to its temporal sensitivity. Specifically, consider

two applications, each registering 1,000 downloads—if one was released a year ago while the other has been available for a decade, equating their popularity would be misleading. This disparity arises because download-based popularity measures are intrinsically dependent on the age of the application. To empirically substantiate this hypothesis, we compute Kendall’s tau correlation coefficient between application age and download count. The analysis yields a correlation coefficient of 0.31, accompanied by a p-value below the conventional significance threshold of 0.05. These results provide statistically significant evidence that download count is positively correlated with application age. For this reason, we collected the apps’ release date and calculated each app’s age. This way, we normalized the age factor and calculated *DownloadsPerYear*.

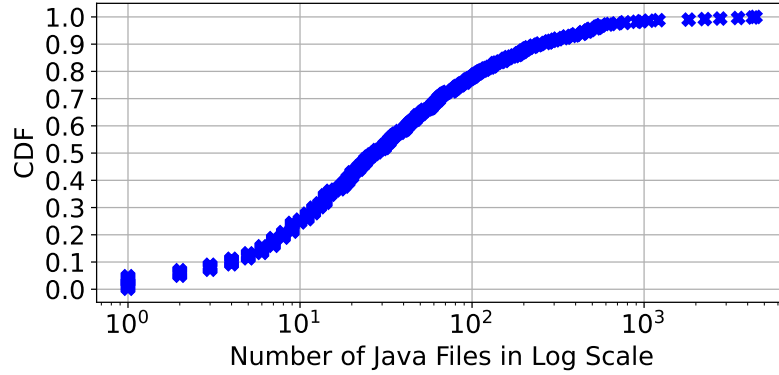
We also discarded apps that are less than 1 year old to enable robust analysis. After this step, we were left with 503 apps. To provide an overview of the dataset, Figure 4.2 presents the cumulative distribution functions (CDFs) for the size of the applications, measured in lines of code (LOC), and the number of Java files per application, showing that our dataset covers apps with very different sizes (small to large). We found that the few very small apps with less than 10 LOC are actually language extension packs. However, these apps are not included in our final dataset (discussed later).

4.3.3 Feature Extraction

We refer to the analysis of source code and the collection of various quality indicators as feature extraction. To gather features that represent the quality of different



(a) CDF of size (LOC).



(b) CDF of number of Java files.

Figure 4.2: Cumulative distribution functions for (a) the size (LOC) of the applications in log scale and (b) the number of Java files per application in log scale.

aspects of an app, we consider three types of features: code metrics, code smells, and metadata. We deliberately exclude platform dependency metrics, device attributes, and performance metrics. Collecting device attributes and performance metrics would necessitate installing and running apps on real devices to gather data, limiting our analysis to a small number of apps and devices. Regarding platform dependency metrics, Syer et al. [220] demonstrated that defect proneness is associated with plat-

form dependency, specifically the extent to which code relies on Android APIs, but we instead focus on code metrics and design issues that indicate defect proneness. Additionally, managing the ever-evolving Android APIs for all of our collected apps is not feasible.

System-Level Metrics. Architectural erosion—stemming from poor modularity and unmanaged dependencies—can significantly increase maintenance overhead and degrade software quality over time. To systematically quantify architectural modularity and assess design-time maintainability risks, we employ the DV8 tool [35], which encapsulates decades of research in software design analysis and education [34].

DV8 is a static analysis framework developed to evaluate software architecture through a suite of metrics grounded in design rule theory. It provides actionable insights by identifying architecture anti-patterns and computing maintainability metrics that span structural, evolutionary, and modularity dimensions. The tool supports architecture-level assessment at scale and across programming languages, making it particularly suited for analyzing large software systems in industrial or open-source contexts.

In our study, DV8 was used to extract a total of 17 features that reflect the architectural health of the target software. These include nine quantitative metrics that capture key maintainability dimensions—such as coupling, change impact, and component autonomy—as well as eight anti-pattern indicators that denote architectural flaws observed in the system. Table 4.1 presents an overview of these 17 features, grouped into two categories: architecture maintainability metrics and anti-pattern counts.

Table 4.1: Seventeen System-Level Metrics Extracted Using the DV8 Tool.

Metric			Description
<i>Architecture Maintainability Metrics</i>			
Decoupling Level (DL)			Degree to which modules are decoupled, reflecting modularity and independence.
DL	Exclude	Isolated	Decoupling level computed while excluding isolated modules.
Items			
Number of Isolated			Penalty score reflecting the impact of isolated modules on decoupling level.
Items DL Score			
Number of Files			Total number of source files analyzed in the system.
Propagation Cost (PC)			Ratio of modules affected by a change, based on dependency propagation analysis.
PC	Exclude	Isolated	Propagation cost excluding isolated modules from the analysis.
Items			
Number of Isolated			Weighted score indicating the effect of isolated modules on propagation cost.
Items Score PC			
Independence Level (IL)			Average structural independence level of all modules in the system.
Number of Isolated			Score capturing the influence of isolated modules on independence level.
Items Score IL			

Continued on next page

Table 4.1: Seventeen System-Level Metrics Extracted Using the DV8 Tool (Continued).

Metric	Description
<i>Architecture Anti-pattern Counts</i>	
Clique Count (Total)	Total instances of strongly connected components.
Clique File Count	Number of files involved in cliques.
Unhealthy Inheritance Count (Total)	Total cases of inheritance-based design smells.
Unhealthy Inheritance File Count	Files affected by inheritance anti-patterns.
Package Cycle Count (Total)	Instances of cyclic dependencies across packages.
Package Cycle File Count	Files implicated in package-level cycles.
Total Anti-patterns Count	Aggregate number of anti-pattern occurrences.
Total Anti-pattern Files	Aggregate file count affected by anti-patterns.

Class-level and method-level metrics. As Java is an object-oriented programming language, we evaluate code quality using the renowned Chidamber and Kemerer (CK) metrics suite. To support this analysis, we employ the CK tool developed by Aniche [14], which computes a comprehensive set of 49 class-level and

29 method-level metrics. Among these are the six canonical CK metrics: Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Objects (CBO), Response for a Class (RFC), and Lack of Cohesion in Methods (LCOM). A detailed overview of some of the selected class-level metrics is provided in Table 4.2. Our shared dataset contains the whole list.

Table 4.2: Class-Level Metrics and Descriptions.

Metric	Description
CBO	Coupling Between Objects: number of classes to which a class is coupled.
WMC	Weighted Methods per Class: sum of McCabe complexity of methods.
DIT	Depth of Inheritance Tree: depth of class in inheritance hierarchy.
NOC	Number of Children: number of immediate sub-classes.
RFC	Response For a Class: number of unique methods that can be invoked.
LCOM	Lack of Cohesion of Methods: dissimilarity of methods in a class.
TCC	Tight Class Cohesion: ratio of directly connected method pairs.

Continued on next page

Table 4.2: Class-Level Metrics and Descriptions (Continued).

Metric	Description
LCC	Loose Class Cohesion: ratio of direct and indirect method connections.
Total Methods Quantity	Total number of methods in a class.
Static Methods Quantity	Number of static methods.
Public Methods Quantity	Number of public methods.
Private Methods Quantity	Number of private methods.
Protected Methods Quantity	Number of protected methods.
Default Methods Quantity	Number of default (package-private) methods.
Visible Methods Quantity	Number of non-private methods.
Abstract Methods Quantity	Number of abstract methods.
Final Methods Quantity	Number of final methods.
Total Fields Quantity	Total number of fields.

Continued on next page

Table 4.2: Class-Level Metrics and Descriptions (Continued).

Metric	Description
Protected Fields Quantity	Number of protected fields.
Default Fields Quantity	Number of default (package-private) fields.
Final Fields Quantity	Number of final fields.
NOSI	Number of Static Invocations: calls to static methods.
LOC	Lines of Code: non-empty, non-comment lines.
Return Quantity	Number of return statements.
Loop Quantity	Number of loop statements (for, while, do-while).
Comparisons Quantity	Number of comparison operations (==, !=).
Try-Catch Quantity	Number of try-catch blocks.
Parenthesized Expressions Quantity	Number of expressions in parentheses.
Assignments Quantity	Number of assignment operations.
Math Operations Quantity	Number of mathematical operations.
Variables Quantity	Number of variable declarations.
Max Nested Blocks Quantity	Maximum depth of nested blocks.

Continued on next page

Table 4.2: Class-Level Metrics and Descriptions (Continued).

Metric	Description
Anonymous Classes	Number of anonymous inner classes.
Quantity	
Inner Classes Quantity	Number of named inner classes.
Lambdas Quantity	Number of lambda expressions.
Unique Words Quantity	Number of distinct words in source code.
Modifiers	Number of modifiers used in the class and its members.

In addition to the 29 method-level metrics provided by the CK tool, we extend the metric set by incorporating an additional metric, Readability, which estimates the cognitive effort required to understand a method based on syntactic and lexical features such as identifier clarity, control structure complexity, and textual coherence. We have used the readability score generation tool proposed by Buse et al. [33]. Conversely, we exclude the *hasJavaDoc* metric from our analysis. This metric simply captures whether a method has associated documentation, without offering any insight into the quality or content of that documentation. Also, as we aggregate all method-level features to produce a fixed-length representation for each app—as described later in this section—this binary metric is not suitable. Its boolean nature prevents meaningful aggregation using statistical measures such as minimum, maximum, mean, or percentiles. A sample set of the 29 method-level metrics used in this study is summarized in Table 4.3.

Table 4.3: A Subset of Method-Level Metrics and Descriptions.

Metric	Description
Fan-In	Number of other methods that call this method.
Fan-Out	Number of methods this method calls.
LOC	Lines of Code: number of lines in the method excluding comments and blanks.
Return Quantity	Number of return statements in the method.
Variables Quantity	Number of variables declared in the method.
Parameters Quantity	Number of parameters the method accepts.
Methods Invoked Quantity	Number of distinct methods invoked.
Methods Invoked Local Quantity	Number of local methods directly invoked.
Methods Invoked Indirect Local Quantity	Number of local methods indirectly invoked.
Loop Quantity	Number of loop constructs in the method (for, while, do-while).
Comparisons Quantity	Number of comparison operations (==, !=, <, >, etc.).
Try-Catch Quantity	Number of try-catch blocks.

Continued on next page

Table 4.3: A Subset of Method-Level Metrics and Descriptions (Continued).

Metric	Description
Parenthesized Expressions Quantity	Number of expressions enclosed in parentheses.
Assignments Quantity	Number of assignment statements.
Math Operations Quantity	Number of mathematical operations (+, -, *, /, etc.).
Max Nested Blocks Quantity	Maximum depth of nested code blocks.
Lambdas Quantity	Number of lambda expressions.
Unique Words Quantity	Number of unique words used in the method source.
Modifiers	Access and non-access modifiers applied to the method.
Log Statements Quantity	Number of logging statements.
Readability	Readability score indicating ease of understanding the method.

While system-level metrics yield a single, fixed set of values per application, class-level and method-level metrics produce variable-sized metric sets depending on the number of classes and methods in each app. However, statistical and machine learning models require a consistent, fixed-length representation across all samples. A common approach to address this involves computing summary statistics such as the minimum,

maximum, mean, and median. Yet, these aggregates are often sensitive to outliers, which can distort the underlying distribution.

To capture the structural patterns more robustly and mitigate the influence of extreme values, we compute 11 percentiles for each metric, ranging from the 10th to the 99th percentile (i.e., 10th, 20th, ..., 90th, 95th, and 99th percentiles). For class-level metrics, we exclude the features *TCC*, *LCC*, and *LCOM** due to their high rate of missing values. It is also important to note that the CK tool's class-level metrics are computed over all types of classes, including normal, anonymous, and inner classes.

For normal classes, we compute the minimum, maximum, mean, and 11 percentiles for all class-level features. In addition to these statistics, some metrics carry global significance. We identify seven such important global metrics: `lambdasQty`, `totalMethodsQty`, `innerClassesQty`, `LOC`, `anonymousClassesQty`, `loopQty`, and `uniqueWordsQty`. For these metrics, we also calculate their total count across the entire codebase for all types of classes to capture their overall impact. To complement the aggregated features, we introduce a new metric, `TotalNormalClasses`, which captures the number of standard classes in the app. Applications with fewer than five normal classes are excluded (57 applications) from the analysis to ensure statistical reliability. This exclusion process removed the small language extension pack apps that we discussed earlier in Section 4.3.2.

For method-level metrics, we exclude constructor methods and compute the minimum, maximum, mean, and the same 11 percentiles for each metric. After performing this aggregation process, we obtain a class-level metrics dataset comprising 652 features and a method-level metrics dataset with 406 features per application.

Code Smells. Since code smells significantly increase the bug proneness of source code [178], we extracted various types of code smells as features as presented in Table 4.4.

Table 4.4: A Subset of Code Smells and Descriptions.

Metric/Smell	Description
Complex Conditional	Presence of nested or convoluted conditional logic that reduces code readability and maintainability.
Complex Method	A method exhibiting high cyclomatic complexity, making it difficult to test and understand.
Empty catch clause	A catch block that does not contain any handling logic, which may hide potential exceptions.
Long Identifier	Overly verbose identifier names that can reduce code clarity.
Long Method	A method that is excessively long, violating the single-responsibility principle.
Long Statement	A single line of code that contains too many operations, reducing readability.
Magic Number	Usage of unexplained numeric literals instead of named constants.
Missing default	A switch-case structure without a default case, risking unhandled conditions.
Deficient Encapsulation	Exposure of internal class details that should be hidden, violating encapsulation principles.
Insufficient Modularization	Modules or classes that are too large or do not encapsulate coherent responsibilities.

Continued on next page

Table 4.4: A Subset of Code Smells and Descriptions (Continued).

Metric/Smell	Description
Unutilized Abstraction	Abstract classes or interfaces that are defined but never instantiated or extended.
Feature Concentration	Multiple unrelated responsibilities implemented within a single class or module.
Long Parameter List	Methods that accept too many parameters, increasing complexity and decreasing reusability.
Broken Hierarchy	Inheritance structures that are incorrectly implemented, leading to poor reuse and maintainability.
Broken Modularization	Logical modules that are fragmented across packages or files.
Cyclic Dependent Modularization	A cycle of module dependencies, leading to tight coupling and hindered modularity.
Imperative Abstraction	Classes that contain detailed implementation rather than abstract behaviors.
Missing Hierarchy	Expected inheritance hierarchies are absent, possibly indicating poor reuse.
Multipath Hierarchy	A class inherits the same superclass through multiple paths, complicating inheritance resolution.
Rebellious Hierarchy	Subclasses that override parent behaviors in ways that break substitutability.
Unexploited Encapsulation	Failure to make use of available encapsulation mechanisms, e.g., public fields.
Unnecessary Abstraction	Abstractions (interfaces or abstract classes) that do not provide additional value.

Continued on next page

Table 4.4: A Subset of Code Smells and Descriptions (Continued).

Metric/Smell	Description
God Component	A module that centralizes too many responsibilities, similar to a God Class.

There are many tools available for code smell detection, some of which take the APK as input [100, 216], while others operate solely on source code [176, 162, 205]. For our analysis, we chose the DesigniteJava tool, which is a version of the original Designite tool [205] specifically designed for Java. The primary reason for selecting this tool is its comprehensive support for various types of code smells, including architectural, design, implementation, and test smells. For this study, we focused on architectural, design, and implementation smells, which collectively account for a total of 34 code smells.

Other Metadata. We identified and collected the following app-specific features: *activity count*, *permissions*, *advertisement support*, and *genre*.

The *activity count* feature represents the number of Android Activities in an app, and thus, is likely to be correlated with the number of features offered by an app. It was calculated by parsing the `AndroidManifest.xml` file of each app.

The data for permissions, advertisement support, and genre were collected from the Google Play Store. The permissions feature is a list of strings, such as Microphone, Location, Contact, etc. Advertisement support indicates whether the app shows advertisements or not.

Genre denotes the application category, and each application is assigned to ex-

actly one genre. The distribution of apps for different genre is presented in Figure 4.3. While our distribution differs from those reported in prior Google Play-only studies [71, 37, 227], it is consistent with Grano et al. [87], who similarly analyzed a dataset constructed by combining Google Play and F-Droid. Notably, several genres, particularly *TOOLS* and *PRODUCTIVITY*, remain prominent across all datasets.

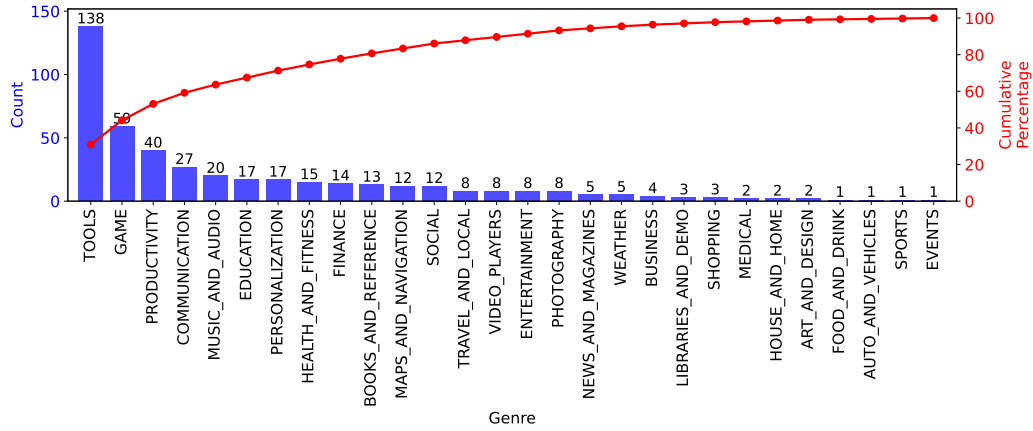


Figure 4.3: Distribution of apps by *genre*, with counts shown as bars and the cumulative percentage indicated by the line. For clarity, the 13 game-related genres are aggregated into a single *GAME* category.

The *activity count* is represented by integer numbers and *advertisement support* is represented as a binary value (indicating the presence or absence of ads), while *genre* and *permissions* are categorical string features. To make these string-based features suitable for machine learning models, we applied one-hot encoding using scikit-learn [184]. This technique transforms each unique category into a separate binary column, where a value of 1 indicates the presence of that category and 0 indicates its absence. This allows models to interpret categorical data without assuming any ordinal relationship between the categories [252].

After processing, we obtained 16 permission features and 40 genre features, resulting in a total of 58 app-specific features for analysis. The features are presented in Table 4.5.

Table 4.5: Metadata Description of Selected App Features.

Feature	Description
Genre	A categorical variable representing the content classification of an application. The possible values include: <i>ART_AND_DESIGN</i> , <i>AUTO_AND_VEHICLES</i> , <i>BOOKS_AND_REFERENCE</i> , <i>BUSINESS</i> , <i>COMICS</i> , <i>COMMUNICATION</i> , <i>DATING</i> , <i>EDUCATION</i> , <i>ENTERTAINMENT</i> , <i>EVENTS</i> , <i>FINANCE</i> , <i>FOOD_AND_DRINK</i> , <i>GAME_ACTION</i> , <i>GAME_ADVENTURE</i> , <i>GAME_ARCADE</i> , <i>GAME_BOARD</i> , <i>GAME_CARD</i> , <i>GAME_CASINO</i> , <i>GAME_CASUAL</i> , <i>GAME_EDUCATIONAL</i> , <i>GAME_PUZZLE</i> , <i>GAME_RACING</i> , <i>GAME_ROLE_PLAYING</i> , <i>GAME_SIMULATION</i> , <i>GAME_SPORTS</i> , <i>GAME_STRATEGY</i> , <i>GAME_TRIVIA</i> , <i>GAME_WORD</i> , <i>HEALTH_AND_FITNESS</i> , <i>HOUSE_AND_HOME</i> , <i>LIBRARIES_AND_DEMO</i> , <i>LIFESTYLE</i> , <i>MAPS_AND_NAVIGATION</i> , <i>MEDICAL</i> , <i>MUSIC_AND_AUDIO</i> , <i>NEWS_AND_MAGAZINES</i> , <i>PARENTING</i> , <i>PERSONALIZATION</i> , <i>PHOTOGRAPHY</i> , <i>PRODUCTIVITY</i> , <i>SHOPPING</i> , <i>SOCIAL</i> , <i>SPORTS</i> , <i>TOOLS</i> , <i>TRAVEL_AND_LOCAL</i> , <i>VIDEO_PLAYERS</i> , and <i>WEATHER</i> .
Continued on next page	

Table 4.5: Metadata Description of Selected App Features (Continued).

Feature	Description
Uses Permissions	A multi-label categorical variable indicating system-level permissions requested by the application. The enumerated values are: <i>Location, Phone, Photos/Media/Files, Storage, Wi-Fi connection information, Device ID & call information, Other, Uncategorized, Camera, Microphone, Identity, Calendar, Contacts, Device & app history, SMS</i> , and <i>Wearable sensors/Activity data</i> . These permissions are indicative of the app’s access scope and potential privacy implications.
Advertisement Support	A binary feature indicating whether the application contains embedded advertising services.
Activity Count	A numerical feature capturing the total number of declared Android activity components in the app’s manifest file.

4.3.4 Feature Selection

After excluding 57 applications from the initial pool of 503 during class-level code metrics extraction, we obtained a final dataset of 446 Android applications, each characterized by 1,167 features. While this comprehensive feature set was designed to capture a wide range of structural and semantic properties of software systems, the inclusion of high-dimensional feature spaces can adversely affect the performance of machine learning models—a phenomenon commonly attributed to the curse of dimensionality. To mitigate this issue and enhance model generalizability, we applied feature selection techniques to reduce the dimensionality of the dataset while preserving relevant information. The specific subset of features employed in each ex-

perimental setting, along with their respective contributions to model performance, is discussed in detail in Section 4.4.

Feature Set 1: Size-only. Although numerous software metrics have been proposed and empirically evaluated as predictors of maintainability and other quality attributes, the reliability of these metrics, beyond basic size, remains a topic of ongoing debate [50, 83, 208, 82]. Several studies argue that the size of code is the most robust and consistent predictor of maintenance effort [67, 83], while many other code metrics exhibit strong collinearity with size [102]. Based on this evidence, we constructed a minimal baseline model using only a size-related feature to assess its standalone predictive power. To measure size, we used the *CLOC* [63] tool, which analyzes a project directory and reports the number of blank lines, comment lines, and lines of code. For each app, we computed the lines of code as size, excluding blank lines and comments.

Feature Set 2: Handpicked. To construct a compact yet representative subset of features, the first and last authors manually selected 28 features based on domain knowledge and prior empirical studies. This expertise-guided selection approach was motivated by the observation that expert-guided feature selection can often outperform traditional algorithm-based feature selection approaches [157, 36]. The first author is a graduate student in Software Engineering with over two years of industry experience, while the last author is an Assistant Professor of Computer Science specializing in Software Engineering, with significant prior work on code metrics [4, 47, 50, 52, 51, 53, 151]. The first selected feature is *size*, given its well-documented predictive value [67, 83, 102]. Additionally, we included two system-level

features: *decouplingLevel*, which quantifies the extent to which a software system adheres to modular design principles [160], and *total_InstanceCount*, representing the cumulative count of detected anti-patterns across the codebase. Together, these features offer a coarse-grained overview of architectural quality.

At the class level, we selected three canonical metrics from the CK suite: Coupling Between Objects (CBO), Weighted Methods per Class (WMC), and Response for a Class (RFC). These metrics have demonstrated consistent relevance in empirical studies evaluating code quality and defect proneness [212, 206, 172, 190]. As adding all the 11 percentiles for these three metrics would increase the feature set size significantly, we computed only the 10th, 50th, and 90th percentiles, resulting in a total of nine class-level features. While the 50th percentile represents the median value, the 10th and 90th percentiles capture values at opposite ends of the distribution.

From the method-level metrics, we selected four: *McCabe*, *fanin*, *fanout*, and *readability*. The *McCabe* metric was selected because of its wide adoption in software quality and maintenance research [60, 130, 223, 200]. *fanin* and *fanout* show the importance of a method and the dependency of a method on other code components, respectively [182, 50]. *readability* metric is of particular interest as it estimates the cognitive load required to comprehend a method, based on syntactic and lexical attributes.

Percentiles (10th, 50th, and 90th) were again computed for each, producing 12 method-level features in total. In addition to these metrics, we incorporated three widely studied code smells: *God Components*, *Long Statement*, and *Long Method*, which have been shown to correlate with software defects and maintainability is-

sues [175, 39, 177]. Finally, we included the *containsAds* feature, a binary indicator of whether the application integrates advertising-related code.

This manually curated feature set balances interpretability and predictive relevance, leveraging insights from prior literature and expert judgment to represent core aspects of software complexity, modularity, and maintainability. However, hand-picked features are inherently subjective and may be influenced by researcher bias. To mitigate this limitation and ensure a more systematic and data-driven selection process, we also employ an automated feature selection approach, described next.

Feature Set 3: Voting. In contrast to the previous two feature sets, we also add another feature set with a more systematic approach by applying feature selection algorithms and voting. Feature selection algorithms can be broadly categorized into filter, wrapper, embedded, hybrid, and ensemble model-based methods [20]. Each of these techniques has its strengths and weaknesses. Li et al. [137] described various scenarios and the appropriate feature selection techniques for each. They recommended an ensemble approach combining multiple types of feature selection techniques for datasets with high dimensionality and small sample sizes. Following their guidance, we employed a combination of feature selection methods, including two filter-type algorithms, one wrapper algorithm, and three embedded-type algorithms.

For classification and regression tasks, the choice of feature selection algorithms was tailored to address the unique characteristics of each problem. For classification, the algorithms were categorized into three types: filter methods, which included Pearson Correlation and Chi2 Correlation; wrapper methods, represented by the Support Vector Classifier; and embedded methods, consisting of Logistic Regression, Random

Forest Classifier, and Light Gradient Boosting Classifier. For regression tasks, a similar categorization was followed. The filter methods included Pearson Correlation and Analysis of Variance, while the wrapper approach utilized the Support Vector Regressor. The embedded methods comprised Lasso Regressor, Ridge Regressor, and Random Forest Regressor.

Each of the feature selection algorithms provides a ranked list of n features, where n represents the number of features selected by the algorithm. For our experiments, we set n to 25, allowing each algorithm to suggest its top 25 features. To aggregate the features suggested by different algorithms, we apply a voting mechanism. Specifically, we select only those features that are recommended by at least 50% of the algorithms, which in our case corresponds to three feature selection algorithms out of the six used. This aggregation ensures that the final selected features are robust and agreed upon by multiple algorithms, reducing the likelihood of including irrelevant or redundant features.

4.3.5 Learning Algorithms and Evaluation Metrics

Encouraged by similar previous studies [47, 94, 68, 17], we employed a set of popular machine-learning algorithms for both classification and regression tasks. For the classification task, we employed a diverse set of algorithms encompassing both simple and complex models. Specifically, we utilized Logistic Regression (LR), Decision Tree Classifier (DT), Random Forest Classifier (RF), Gradient Boosting Classifier (GB), and Multi-Layer Perceptron (MLP). This selection spans a spectrum of model complexities, ranging from simple and linear approaches such as LR to highly non-linear

and complex models like MLP. To evaluate the performance of these models, we calculated class-wise metrics, including precision, recall, F1 score, area under the curve (AUC), and Matthews Correlation Coefficient (MCC).

For the regression tasks, we employed a diverse suite of models, including Lasso, Ridge, Decision Tree Regressor (DT), Random Forest Regressor (RF), Gradient Boosting Regressor (GB), and Multi-Layer Perceptron Regressor (MLP). The selection strategy mirrors that of the classification setting, encompassing both linear models (e.g., Lasso, Ridge) and more complex non-linear estimators (e.g., MLP). The regression models were evaluated using metrics such as root mean squared error (RMSE), mean absolute error (MAE), and the coefficient of determination (R^2 score).

To ensure robust and unbiased model evaluation, we employed the Leave-One-Out Cross-Validation (LOOCV) technique. LOOCV is a reliable method for small datasets, where each instance serves as a test set exactly once while the remaining instances are used for training. By aggregating the results across all iterations, we obtained a comprehensive understanding of the models' generalization performance.

Parameter and hyperparameter optimization techniques are computationally expensive [26], particularly when combined with LOOCV, as they would require us to retrain each model for each application. Therefore, consistent with prior studies [47, 182], all machine learning models in this work were trained using the default configurations provided by their respective libraries. Although we conducted preliminary experiments in which the parameters and hyperparameters for both regression and classification models were randomly varied, these experiments did not yield any performance improvements over the default settings. This observation discouraged

us from trying the grid search approach, which is even more computationally intensive. We acknowledge that future studies can try different settings to evaluate if our reported accuracy can be improved.

4.4 Approach, Analysis, and Results

In this section, we present the approach used to address each of the two research questions, along with the corresponding findings.

4.4.1 RQ1: To what extent can internal software metrics predict an app's user rating?

To address this research question, we first examined the feasibility of predicting actual app ratings. Next, we categorized the apps into two groups, *Popular* and *Unpopular*, based on their ratings, and investigated the feasibility of predicting an app's popularity. To achieve this, we conducted a series of experiments using regression and classification models.

Regression: Predicting the Actual Ratings

User ratings for mobile applications are generally provided on a discrete scale ranging from 1 to 5. To enable a continuous prediction task, we computed the mean rating for each application based on all associated user reviews. Applications with no user reviews are excluded from the analysis, as they cannot have a valid rating assigned (ratings cannot be less than 1). After removing these 39 applications, our

regression analysis is conducted using the remaining 407 applications.

As outlined in Section 4.3.4, we considered three distinct feature sets for our analysis: **Size-only**, **Handpicked**, and **Voting**. Among these, the **Voting** approach to feature selection identified three features as most predictive of user ratings: `fanin_30_class`, `logStatementsQty_max_class`, and `genreId_GAME_BOARD`. For example, `fanin_30_class` is the 30th percentile of the metric `fanin` at the class level granularity.

Building upon these three feature sets, we conducted three groups of experiments in which six regression models were applied to predict the continuous app ratings: *MLPRegressor* (MLP), *Lasso*, *Ridge*, *DecisionTreeRegressor* (DT), *RandomForestRegressor* (RF), and *GradientBoostingRegressor* (GB). To ensure that the predicted ratings remained within the valid interval [1,5], we applied a post-processing step wherein predictions falling below 1 were clamped to 1, and those exceeding 5 were truncated to 5. This normalization was necessary due to the unbounded nature of regression outputs, which can produce values outside the expected domain. The comparative results of the regression models are summarized in Table 4.6.

Overall, the results indicate a clear trend: models trained on the **Voting** feature set consistently outperform those trained on **Size-only** and **Handpicked** features. Models trained using the **Size-only** feature set generally performed worse, with higher RMSE and MAE values and R^2 values that are below zero. This underscores the limited predictive power of size-related metrics alone and highlights the added value of incorporating domain-informed or statistically selected features. Among the evaluated models, Ridge regression achieved the best overall performance using the **Voting** features, with an RMSE of 0.69, MAE of 0.51, and the highest R^2 value of

0.05. In contrast, the *MLPRegressor* demonstrated the weakest performance.

Table 4.6: Regression metrics (RMSE, MAE, R^2) of rating regressors evaluated using three feature selection strategies: Size-only, Handpicked, and Voting. Bold values indicate the best metric scores within each feature set.

Model	Size-only			Handpicked			Voting		
	RMSE	MAE	R^2	RMSE	MAE	R^2	RMSE	MAE	R^2
MLP	3.00	2.90	-16.92	2.10	1.76	-7.82	0.73	0.56	-0.05
Lasso	0.71	0.53	-0.01	0.71	0.53	-0.01	0.71	0.53	0.00
Ridge	0.71	0.53	-0.01	0.72	0.54	-0.03	0.69	0.51	0.05
DT	1.05	0.78	-1.20	1.01	0.76	-1.04	0.73	0.55	-0.07
RF	0.89	0.67	-0.59	0.72	0.55	-0.03	0.72	0.54	-0.03
GB	0.80	0.58	-0.26	0.75	0.57	-0.12	0.71	0.53	0.00

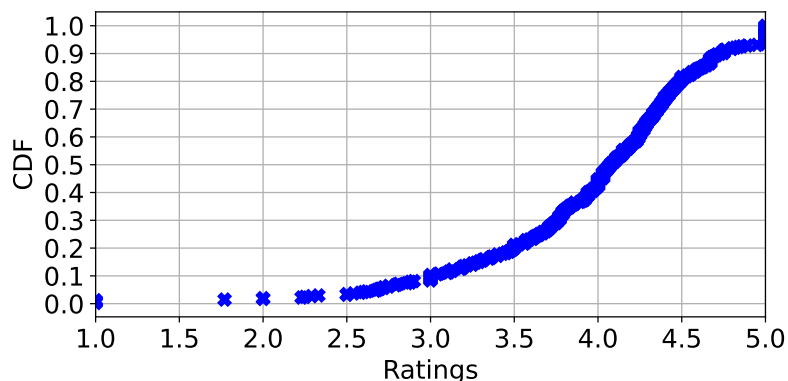
A closer examination of the regression outcomes reveals that, for the majority of cases, the RMSE remains below 1 and the MAE is under 0.6. These values may suggest that the internal code metrics possess reasonable predictive capability: for instance, the Ridge regression model achieves an MAE of 0.51, indicating that it can predict application ratings with around half-point deviation on average. Such results imply that, in absolute terms, the models are able to approximate user ratings with relatively low error.

However, this positive impression is tempered by a contrasting trend in the R^2 scores. While RMSE and MAE reflect the magnitude of prediction errors, the R^2 metric evaluates the proportion of variance in the target variable that is explained

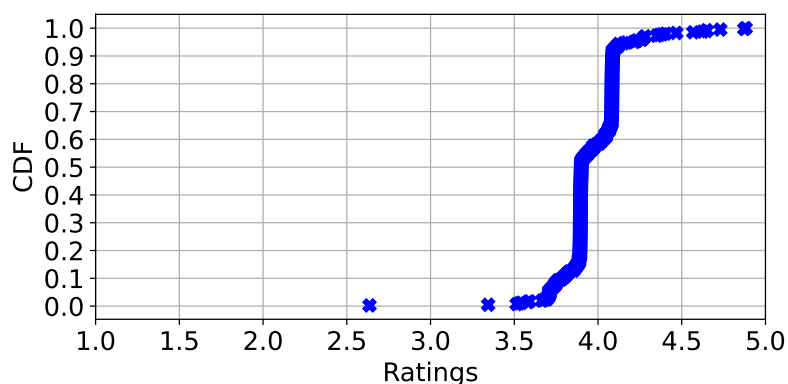
by the model. While R^2 values close to 1 indicate strong explanatory power, the metric can range from $-\infty$ to 1. A value of 0 means the model performs no better than predicting the mean of the target variable, while negative values indicate that the model performs worse than this baseline. In our experiments, most models yield R^2 scores that are near zero or even negative, suggesting limited ability to capture the variance in user ratings. Notably, the Ridge model—despite its low error metrics—achieves only a modest R^2 of 0.05, and several models (e.g., MLP) produce significantly negative values.

To better understand the limited performance, we analyzed the distribution of both actual and predicted ratings using the Cumulative Distribution Function (CDF), as shown in Figure 4.4. The CDF of the true application ratings (Figure 4.4a) reveals a significant skew in the dataset: approximately 10% of applications received rating 3, while over 5% achieved the maximum rating (5). Most notably, nearly 70% of the applications fall within the 3.0 to 4.5 range, indicating a concentrated distribution around moderately high ratings. This skew is reflected in the predicted values generated by the Ridge regression model (Figure 4.4b), where more than 95% of predictions also lie within the 3.0 to 4.5 interval.

The model exhibits a clear bias toward the dominant region of the target distribution, which contributes to the relatively low RMSE and MAE values, as illustrated in Figure 4.5. Approximately 90% of the absolute prediction errors fall below 1.0, aligning with the observed Mean Absolute Error (MAE) of 0.51. However, a small subset of instances incurs substantially higher errors, likely corresponding to outliers whose true ratings deviate significantly from the distribution’s central mass. Notably,



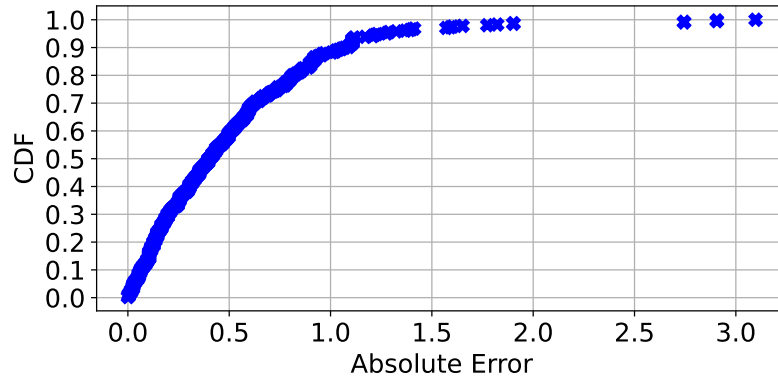
(a) CDF of actual ratings.



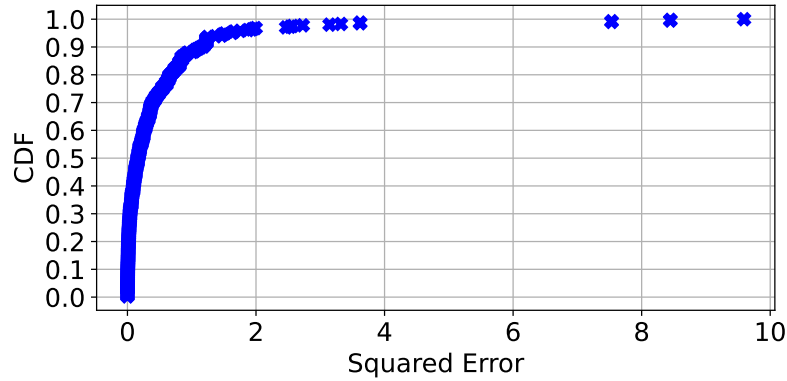
(b) CDF of predicted ratings of Ridge regression model.

Figure 4.4: Cumulative Distribution Functions (CDFs) of actual and predicted ratings. Approximately 70% of the actual ratings fall between 3.0 and 4.5, while over 95% of the predicted ratings lie within this range, indicating the model’s bias towards predicting around the mean value.

while 60% of the original ratings fell between 3.5 and 4.5, the model predicted approximately 95% of applications within this narrow range, indicating a strong centralizing tendency. This behavior reflects the model’s bias towards the most frequent rating region in the training data, causing predictions to cluster around the global mean



(a) CDF of absolute errors.



(b) CDF of squared errors.

Figure 4.5: Cumulative Distribution Functions (CDFs) of absolute error and squared error for the Ridge regression model in predicting actual ratings. The plots indicate that a small number of data points account for the majority of the large errors.

rather than capturing the full range of rating variation.

To further verify this, the predicted ratings were plotted against the true ratings in a scatter plot, as shown in Figure 4.6. Additionally, we included reference lines corresponding to the mean true rating and the bounds defined by $\pm$ the mean absolute error. Most predictions clustered closely around the mean rating line, and nearly all

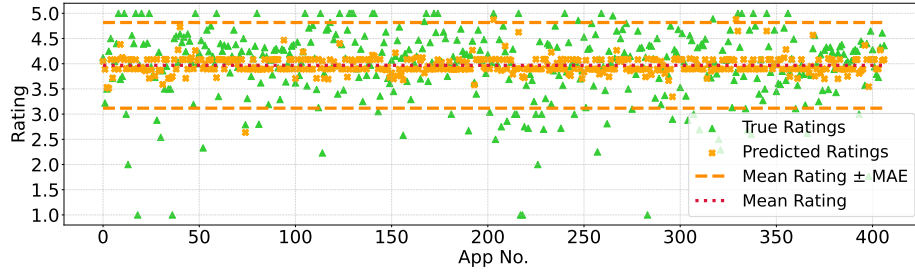


Figure 4.6: Scatter plot of true ratings and predicted ratings for each app.

predicted values fell within the $\pm$ MAE bounds, reinforcing the model's tendency to predict values near the global average.

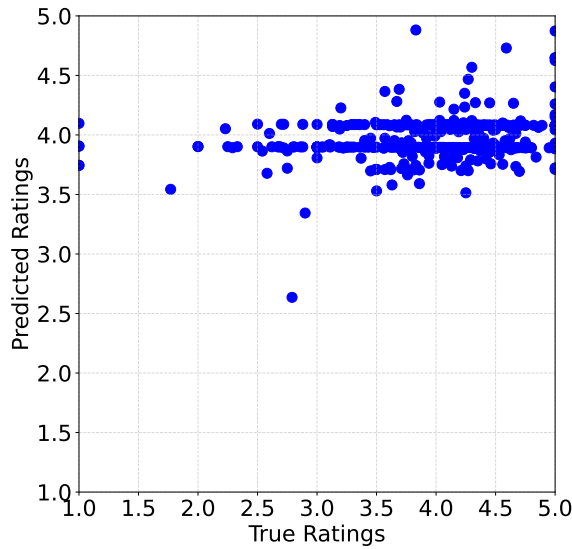


Figure 4.7: Scatter Plot of True Ratings vs Predicted Ratings.

Finally, we assessed the alignment between true and predicted ratings using a standard scatter plot, with the actual ratings on the x-axis and the predicted ratings on the y-axis (Figure 4.7). Ideally, points should concentrate along the diagonal line for an accurate predictive model. However, our results show that predicted ratings were predominantly concentrated between 3.5 and 4.5.

In summary, the strong skew in the dataset leads the regression models to consistently predict values near the mean, resulting in low RMSE and MAE but offering limited insight into the underlying patterns of the data. This behavior is reflected in the uniformly low—often negative— R^2 values, indicating that the models fail to meaningfully capture the variance in user ratings. Consequently, the regression results do not offer conclusive evidence regarding the predictive utility of internal code metrics; they neither confirm their effectiveness nor definitively refute their relevance.

To address the inherent bias introduced by the skewed distribution, we propose an alternative approach: reframing the problem as a classification task by discretizing the continuous rating scores into categorical classes. This transformation may help mitigate the centralizing effect observed in regression and enable the models to better distinguish between applications of varying quality. In the following set of experiments, we explore this classification-based formulation to evaluate its potential in uncovering relationships between internal metrics and ratings.

Classification: Predicting Popularity Based on Ratings.

We began by categorizing apps into *Popular* and *Unpopular* based on user ratings. Apps with ratings ≥ 3.5 were classified as *Popular*, while those with ratings below 3.5 were labeled as *Unpopular*, following thresholds established in prior studies [122, 38]. In this experiment, we retained applications with no user reviews, as the absence of reviews strongly indicates that these applications are indeed *Unpopular*. This classification resulted in a dataset of 327 *Popular* apps and 119 *Unpopular* apps, presenting a noticeable class imbalance.

Similar to the previous experiments, we also have three distinct feature sets for the classification task: **Size-only**, **Handpicked**, and **Voting**. The **Voting** method highlighted five features: `genreId_TRAVEL_AND_LOCAL`, `containsAds`, `parametersQty_20`, `parametersQty_99`, and `innerClassesQty_50_class`. These feature sets were used to train and evaluate various classification models using Leave-One-Out Cross-Validation (LOOCV), as outlined in Section 4.3.5.

To address the inherent class imbalance in our dataset, we implemented three widely used resampling techniques: Random Over Sampling (ROS), Random Under Sampling (RUS), and Synthetic Minority Oversampling Technique (SMOTE) [40]. These techniques were applied to ensure that the classifier had a more balanced representation of different classes, ultimately improving model generalization. Given that we employed LOOCV as our validation strategy, resampling was performed within each fold to prevent data leakage and maintain the integrity of the training process. This ensured that the model was evaluated on unseen data without exposure to artificially generated or resampled instances from the test fold. Additionally, the numerical features were normalized by applying standard scaling using scikit-learn [184], which standardizes each feature by removing its mean and scaling it to unit variance. This normalization is particularly beneficial for models that rely on gradient-based optimization, such as neural networks.

Table 4.7: Performance metrics (Macro F1, MCC, AUC) of rating-based popularity classifiers evaluated using three feature selection strategies: Size-only, Handpicked, and Voting. Bold values indicate the best metric scores within each feature set.

Model	Size-only			Handpicked			Voting		
	Macro F1	MCC	AUC	Macro F1	MCC	AUC	Macro F1	MCC	AUC
MLP	0.42	0.00	0.49	0.56	0.12	0.60	0.58	0.27	0.63
LR	0.42	0.00	0.37	0.58	0.18	0.63	0.56	0.23	0.62
DT	0.51	0.02	0.51	0.53	0.07	0.54	0.55	0.15	0.56
RF	0.52	0.04	0.52	0.53	0.17	0.57	0.55	0.16	0.57
GB	0.52	0.09	0.51	0.56	0.19	0.55	0.56	0.21	0.56
MLP+SMOTE	0.50	0.16	0.63	0.59	0.18	0.62	0.72	0.45	0.72
LR+SMOTE	0.41	0.10	0.60	0.60	0.21	0.64	0.55	0.20	0.73
DT+SMOTE	0.46	-0.04	0.48	0.51	0.03	0.52	0.50	0.07	0.63
RF+SMOTE	0.46	-0.04	0.50	0.59	0.18	0.59	0.52	0.11	0.64
GB+SMOTE	0.54	0.09	0.52	0.58	0.16	0.57	0.51	0.13	0.66
MLP+RUS	0.50	0.14	0.63	0.56	0.18	0.62	0.56	0.13	0.61
LR+RUS	0.40	0.09	0.65	0.55	0.15	0.59	0.55	0.21	0.66
DT+RUS	0.50	0.07	0.54	0.46	-0.02	0.49	0.56	0.13	0.61
RF+RUS	0.50	0.06	0.54	0.52	0.10	0.58	0.68	0.39	0.65
GB+RUS	0.51	0.05	0.54	0.51	0.06	0.55	0.66	0.37	0.66
MLP+ROS	0.51	0.16	0.57	0.58	0.16	0.61	0.50	0.04	0.65
LR+ROS	0.40	0.08	0.51	0.61	0.24	0.64	0.55	0.20	0.71
DT+ROS	0.51	0.02	0.51	0.56	0.11	0.56	0.47	-0.00	0.57
RF+ROS	0.51	0.02	0.51	0.55	0.14	0.58	0.47	-0.02	0.59
GB+ROS	0.55	0.10	0.53	0.56	0.12	0.55	0.48	0.01	0.58

The results of our classification experiments, presented in Table 4.7, mirror trends

observed in the regression setting: models trained on the **Voting** feature set consistently outperform those utilizing either the **Size-only** or **Handpicked** feature sets. In particular, models trained solely on the **Size-only** features exhibit the weakest performance, with MCC values that are close to or below zero. This indicates poor discriminative ability, often comparable to or worse than random guessing.

Among the resampling strategies evaluated, the combination of *MLPClassifier* with Synthetic Minority Oversampling Technique (SMOTE) yielded the most substantial performance gains. Compared to other approaches, such as Random Oversampling (ROS) and Random Under Sampling (RUS), SMOTE proved more effective by generating synthetic instances of the minority class rather than merely duplicating existing samples or removing majority class instances. This synthetic data augmentation enabled the classifier to better approximate the decision boundary between classes, thereby enhancing the model's generalization and robustness.

Overall, the *MLPClassifier* trained with SMOTE on the **Voting** feature set achieved the strongest and most balanced performance across all evaluated metrics. The model attained a Macro F1 score of 0.72, indicating a balanced classification performance across both classes. Furthermore, the MCC increased to 0.45, indicating a meaningful improvement in overall predictive reliability and suggesting that the comparatively high F1 score was not due to chance. The Area Under the Receiver Operating Characteristic Curve (AUC-ROC) also reached 0.72, suggesting that the model was more capable of distinguishing between the positive and negative classes.

The class-wise performance of the SMOTE-enhanced *MLPClassifier*, shown in Table 4.8, highlights its effectiveness in mitigating class imbalance. The model achieved

an F1 score of 0.601 for the *Unpopular* class, indicating improved minority class recognition while still facing some misclassification challenges. Meanwhile, the *Popular* class demonstrated stronger performance with a precision of 0.862, a recall of 0.823, and an F1 score of 0.842, ensuring high reliability in identifying high-rated apps. These results confirm that SMOTE effectively improved recall for the minority class while maintaining high precision for the majority class, leading to a Macro F1 score of 0.72 and MCC of 0.45. By generating synthetic examples, SMOTE allowed the model to learn better decision boundaries, enhancing overall classification performance. The confusion matrix and AUC-ROC curve in Figure 4.8 further illustrate the classifier’s improved ability to separate *Popular* from *Unpopular* apps, as reflected in a higher true positive rate for the minority class and an AUC score of 0.72.

Table 4.8: Classification performance for rating-based popularity prediction using MLP classifier with SMOTE.

Class	Precision	Recall	F1-Score
Unpopular	0.567	0.639	0.601
Popular	0.862	0.823	0.842

While the classification results are encouraging, they are not perfect. The findings suggest that code metrics contribute meaningfully to predicting application popularity, but they are not sufficient in isolation to achieve optimal performance. The moderate classification scores indicate that, although useful, these features capture only a portion of the factors influencing user perception and engagement.

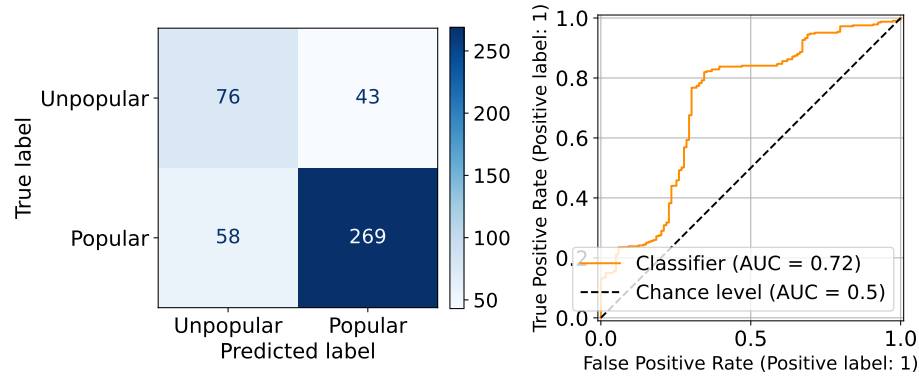


Figure 4.8: Confusion matrix and AUC-ROC curve for rating-based popularity prediction using MLP classifier with SMOTE.

Summary of RQ1 Due to the skewed data, the regression models did not learn much and made predictions around the mean only. As a result, we were unable to draw meaningful conclusions about the usefulness of internal metrics from the regression results alone. In contrast, classification models—especially *MLPClassifier* with SMOTE—demonstrated significantly better performance in distinguishing between *Popular* and *Unpopular* apps, highlighting the predictive value of internal software metrics when the problem is appropriately framed.

4.4.2 RQ2: To what extent can internal software metrics predict an app's number of downloads?

Regression: Predicting the Actual *DownloadsPerYear*.

As we explained in Section 4.3.2, the *DownloadsPerYear* measure was created by normalizing the age impact on total download count. In Figure 4.9, we see the CDF of *DownloadsPerYear* in the log scale, where some values are too high and can be

treated as outliers. Removing these outliers will help us to understand the distribution better.

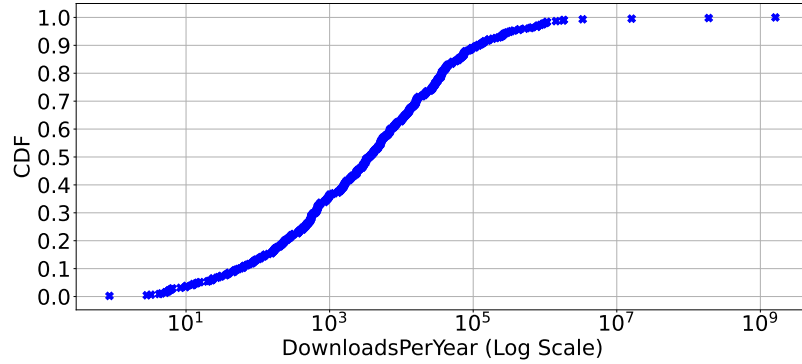


Figure 4.9: CDF of *DownloadsPerYear*.

We apply the Interquartile Range (IQR) method for outlier detection to remove the outliers from the dataset. This approach filters out 65 apps, leaving us with 381 apps for the analysis, with *DownloadsPerYear*. The CDF of *DownloadsPerYear* after the outlier removal is plotted in Figure 4.10.

Next, we have three feature sets similar to the previous RQ. The Voting feature selection method identified nine key features: `maxNestedBlocksQty_70_class`,

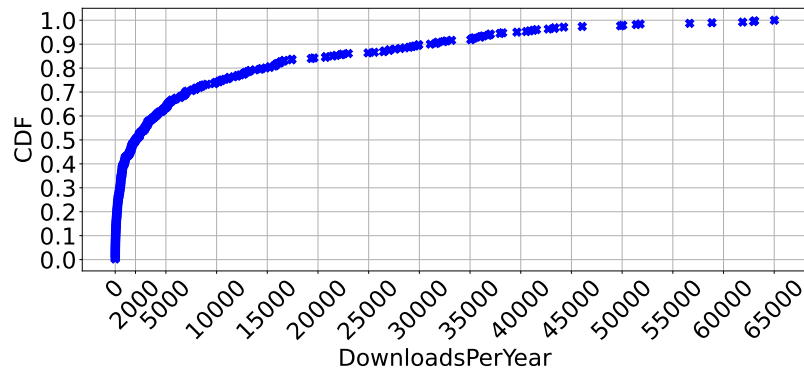


Figure 4.10: CDF of *DownloadsPerYear* after removing outliers.

Table 4.9: Regression metrics (RMSE, MAE, R^2) of *DownloadsPerYear* regressors evaluated using three feature selection strategies: Size-only, Handpicked, and Voting. Bold values indicate the best metric scores within each feature set.

Model	Size-only			Handpicked			Voting		
	RMSE	MAE	R^2	RMSE	MAE	R^2	RMSE	MAE	R^2
MLP	15549.39	8320.82	-0.31	15292.60	8141.81	-0.26	14887.51	7933.33	-0.20
Lasso	13589.72	9876.01	0.00	13957.50	9839.24	-0.05	13031.64	8998.82	0.08
Ridge	13589.72	9876.01	0.00	13981.40	9851.91	-0.06	13030.70	8998.74	0.08
DT	17628.71	11013.86	-0.68	19261.85	12187.56	-1.01	17924.23	10699.23	-0.74
RF	15478.00	10241.84	-0.29	14288.05	10545.20	-0.10	13631.60	9594.51	-0.00
GB	14600.80	9756.98	-0.15	14846.89	10570.74	-0.19	14330.94	9708.88	-0.11

`finalFieldsQty_95`, `Storage`, `wmc_40_class`, `Photos/Media/Files`, `returnQty_30`, `tryCatchQty_max_method`, `lcom_90`, and `fanin_20_class`.

With these feature sets, we apply 6 regression models. As shown in Table 4.9, all models exhibit high error values and low R^2 scores, indicating limited predictive power across the board. Among them, the Ridge regression model using the **Voting** feature set achieved the lowest RMSE (13,030.70) and the highest R^2 (0.08), offering the most consistent performance across all metrics. Although the overall results remain unsatisfactory, we select Ridge regression for further analysis due to its relatively stronger performance and greater stability compared to other models.

The Cumulative Distribution Function (CDF) of Absolute Errors in Figure 4.11 reveals that for 60% of the samples, the absolute error exceeds 5000, which is significantly large in the context of *DownloadsPerYear*. The model achieves an absolute

prediction error below 1,000 for only about 5% of the samples, indicating that accurate predictions are rare and overall predictive accuracy is poor.

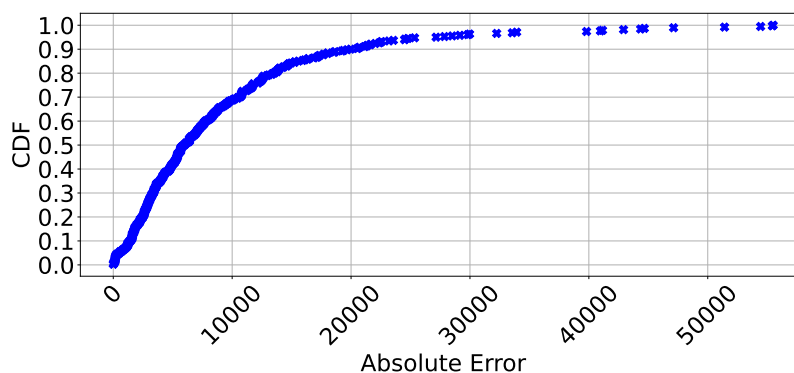


Figure 4.11: CDF of absolute error for the Ridge regression model in predicting *DownloadsPerYear*.

To further examine model performance, we analyze the scatter plot of true versus predicted values in Figure 4.12. In a well-performing regression model, the data points should align closely along the diagonal, indicating accurate predictions. However, as shown in the figure, the predicted values are heavily concentrated below 20,000, despite the true values spanning a much wider range. This compression of output highlights the model’s inability to capture the full variability in *DownloadsPerYear*. These findings reinforce the conclusion that regression models, when relying solely on internal software metrics, perform poorly.

In the previous research question, internal software metrics proved ineffective for regression due to the skewed distribution of app ratings. However, in this research question for *DownloadsPerYear*, the limitation stems from a different issue: the target variable spans a wide range—from 0 to over 65,000—across only 381 apps, making accurate value prediction challenging. Once again, internal metrics failed to provide

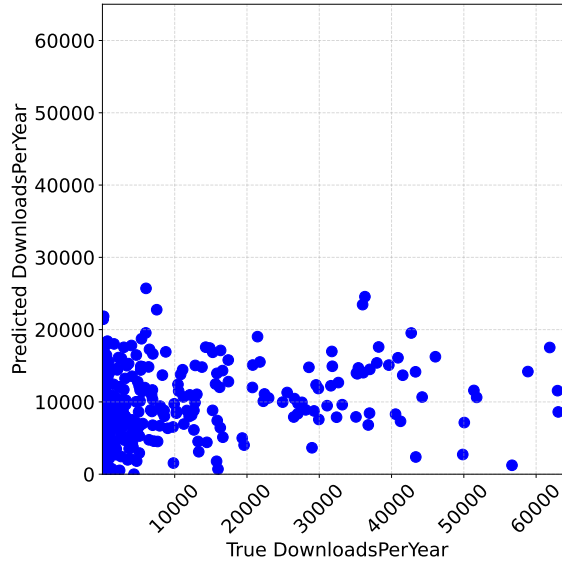


Figure 4.12: Scatter plot of true *DownloadsPerYear* vs predicted *DownloadsPerYear*.

reliable regression performance, prompting us to reframe the problem as a classification task.

Classification: Predicting Popularity Based on *DownloadsPerYear*.

We repeated the classification experiments using *DownloadsPerYear* as the popularity indicator. Based on the cumulative distribution function shown in Figure 4.10, we defined a threshold of 2000 downloads per year, categorizing apps with *DownloadsPerYear* ≥ 2000 as *Popular* and those below 2000 as *Unpopular*. The threshold of 2000 splits the dataset almost exactly in half, with 191 Popular and 190 Unpopular apps.

The Voting method selected five features: `genreId_TOOLS`, `publicFieldsQty_90`, `loc_min_method`, `staticMethodsQty_95`, and `Photos/Media/Files`. With three feature sets: **Size-only**, **Handpicked**, and **Voting**, we train and evaluate the clas-

sification models. As the dataset is already balanced, resampling techniques are not required in this case. Table 4.10 presents the classification results, where the `MLPClassifier` achieved the best overall performance on the `Voting` feature set, with the highest MCC (0.37) and a strong AUC of 0.71. Although other models, such as Logistic Regression, attained slightly higher AUC values on specific feature sets, `MLPClassifier` demonstrated the most balanced performance across all three metrics within the `Voting` configuration.

Table 4.10: Classification metrics (Macro F1 Score, MCC, AUC) of *DownloadsPerYear* classifiers evaluated using three feature selection strategies: Size-only, Handpicked, and Voting. Bold values indicate the best metric scores within each feature set.

Model	Size-only			Handpicked			Voting		
	Macro F1	MCC	AUC	Macro F1	MCC	AUC	Macro F1	MCC	AUC
MLP	0.54	0.13	0.54	0.57	0.14	0.57	0.69	0.37	0.71
LR	0.49	0.09	0.47	0.55	0.10	0.59	0.67	0.33	0.72
DT	0.57	0.14	0.57	0.55	0.11	0.55	0.58	0.16	0.58
RF	0.57	0.15	0.57	0.59	0.19	0.64	0.62	0.25	0.65
GB	0.56	0.11	0.58	0.59	0.18	0.60	0.66	0.32	0.70

The class-wise performance metrics and confusion matrix are presented in Table 4.11 and Figure 4.13, respectively. The model achieves a precision of 0.682 and a recall of 0.689 for the Unpopular category, leading to an F1-score of 0.685. Meanwhile, for the Popular category, the model attains a precision of 0.687 and a recall of 0.680, resulting in an F1-score of 0.684. These results indicate that the model maintains

a balanced performance across both classes, unlike what we observed in RQ1 for popularity based on ratings. Overall, these results again confirm the usefulness of internal metrics.

Table 4.11: Classification performance for *DownloadsPerYear*-based popularity prediction using MLP classifier.

Class	Precision	Recall	F1-Score
Unpopular	0.682	0.689	0.685
Popular	0.687	0.680	0.684

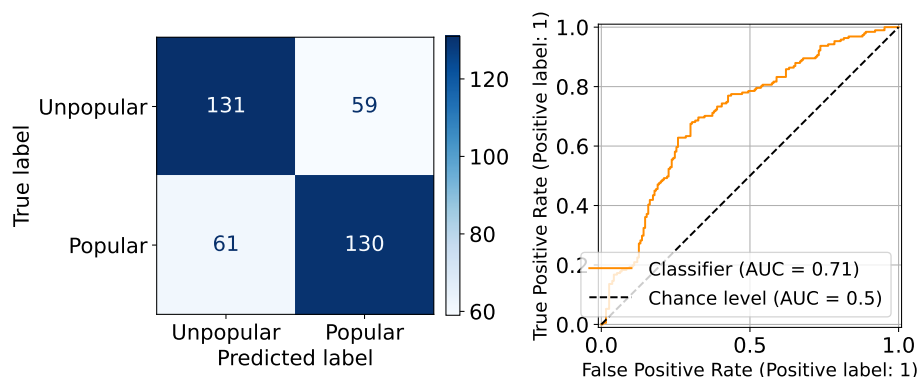


Figure 4.13: Confusion matrix and AUC-ROC curve for *DownloadsPerYear*-based popularity prediction using MLP classifier.

Summary of RQ2 Regression models using internal software metrics struggled to predict *DownloadsPerYear* accurately, with high error rates and low R^2 values, even after outlier removal. Reframing the task as a classification problem improved performance modestly, with the MLP classifier achieving balanced precision and recall, suggesting some potential for distinguishing between popular and unpopular apps.

4.5 Discussion

Our empirical investigation into the predictive power of internal software metrics for application popularity reveals both promising insights and notable limitations. In addressing RQ1—predicting application ratings via regression—we found that the models exhibited limited generalizability, reflected in low R^2 scores, and failed to capture the underlying variability in the data. However, when reframed as a classification task (i.e., distinguishing popular from unpopular apps), model performance improved. These results suggest that internal software metrics carry meaningful predictive value at release time. A similar pattern emerged in RQ2, which examined the prediction of the *DownloadsPerYear* metric.

It is unsurprising that software metrics alone do not fully explain application popularity. Clearly, non-code factors, such as developer reputation, historical performance, the level of competition for the app, the usability of its UI, and marketing efforts, will likely play a significant role in shaping popularity outcomes [222]. Incorporating these features in future work may lead to more accurate and comprehensive predictive models. Considering the potential influence of the non-code factors, it is striking that internal software metrics can still offer valuable insight into app popularity. What seems obvious from this study is that a development organization would be unwise to ignore software metrics in deciding whether a product is ready for release.

Implications for Practitioners

Our findings suggest that different internal metrics can contribute signals for predicting app popularity. Practitioners should therefore view code-level metrics not

only as maintainability indicators but also as potential predictors of market performance. We recommend that software development teams systematically incorporate code quality monitoring into their development processes. The integration of static analysis tools capable of tracking complexity, coupling, and code smells can aid in identifying latent issues that may impact both maintainability and user perception. Early detection of such issues allows teams to implement preventive measures, thereby improving both software quality and user satisfaction and potentially warding off technical debt [34]. Furthermore, practitioners are encouraged to explore the use of machine learning and predictive analytics as part of their quality assurance strategy.

Implications for Researchers

From a research standpoint, our findings provide meaningful insights into the role of internal software metrics in predicting application popularity. Contrary to earlier studies [67, 102, 83, 199] that positioned code size as the principal—or sole—predictor of software quality and performance, our findings reveal that a wider range of metrics also contribute meaningfully to predictive accuracy, especially when context is considered as the king (regression vs. classification), as was also concluded by some other relevant studies—class-level vs method-level evaluation [130, 52], and #revisions vs. change size based evaluation [50].

These findings should encourage the research community to develop a more comprehensive set of code metrics and to evaluate them based on different contexts. Additionally, the improved performance achieved using feature selection algorithms underscores the importance of this preprocessing step in predictive modeling. These

findings should motivate future work on designing and evaluating more sophisticated feature selection algorithms to further enhance prediction performance. We also observed that a model's performance is significantly impacted by the type of algorithm used for addressing the class imbalance problem, indicating a strong need for additional research in this direction. Researchers may also focus on building genre-specific models rather than generic ones, as genre type emerged as an important feature in both of our classification models. However, the small number of apps that are commonly available on both F-Droid and the Google Play Store would be a challenge for building such models.

4.5.1 Threats to Validity

The *external validity* of our study is limited by our focus on Android apps exclusively. As a result, the findings may not be generalizable to iOS apps. Additionally, within the Android ecosystem, we concentrated solely on Java-based applications, although other programming languages, such as Kotlin, are also commonly used to develop Android apps. We can not imagine any reason why the difference in platform or language would significantly affect our findings, but it still remains an external threat.

Also, our dataset is relatively small, as we specifically needed apps available on both F-Droid and the Google Play Store. However, linking these two repositories has been the most viable approach when access to both source code and popularity indicators were required [87, 38, 57].

Internal validity may be affected by the presence of confounding variables, such as

user reviews, marketing efforts, pricing strategies, and external events like app store promotions, which were not fully controlled in our analysis.

Construct validity also poses a challenge, as the operational definitions of “code quality” and “app popularity” may not fully capture their conceptual meanings. While app popularity was measured primarily through downloads and ratings, factors such as user engagement, revenue, and retention rates could offer a more holistic perspective. Similarly, relying on a specific method, such as static code analysis, to measure code quality introduces the risk of mono-method bias, as alternative tools or frameworks might yield different insights.

Conclusion validity of our study is influenced by the various threats outlined above. Although the inclusion of genre-level features provides valuable insights, the findings may not fully account for variations within individual app genres, underscoring the need for more granular analysis to better understand genre-specific patterns.

4.6 Conclusion

This study investigates the relationship between internal software metrics and app popularity indicators, specifically user ratings and download counts. While our results indicate that internal software metrics alone are insufficient for accurately predicting continuous measures such as ratings or yearly downloads, they demonstrate predictive power when app popularity is reformulated as a binary classification problem. These findings highlight the potential utility of internal code metrics in understanding and characterizing software success.

Our study underscores the potential of techniques, such as predictive modeling,

to augment the evaluation and forecasting of app success. These methodologies offer promising directions for identifying critical determinants of performance and fostering innovation in software development practices. For practitioners, our findings advocate for sustainable and quality-driven coding practices aligned with market demands. For researchers, the results affirm the relevance of internal software metrics and encourage further exploration into their role, beyond size, in shaping software outcomes.

In summary, this work contributes to the growing body of research at the intersection of software engineering and market-oriented evaluation, advancing our understanding of how technical quality metrics relate to app success in competitive marketplaces.

Chapter 5

Conclusions and Future Work

This chapter summarizes the contributions of the three studies, discusses what they collectively reveal about software-metric-based prediction, and outlines directions for future research.

5.1 Summary of the Contributions

This thesis investigates how far internal software metrics can take us in predicting future software outcomes. Across three studies, we first improve the data used for method-level prediction, then evaluate method-level bug prediction under realistic conditions, and finally examine whether internal metrics can predict an external, user-facing outcome: app popularity.

Together, the studies show that the usefulness of internal metrics is context-dependent. Their predictive value depends on factors including data quality, prediction granularity, target formulation, class distribution, and evaluation protocol.

5.1.1 Removing Tangled Noise

Method-level bug datasets commonly label every method modified in a bug-fix commit as buggy. However, bug-fix commits are often *tangled*, mixing the actual fix with unrelated changes and thereby introducing incorrect labels.

In Chapter 2, we investigated whether LLMs can determine whether a method-level change contributes to the bug-fixing intent of a commit. Zero-shot LLMs perform effectively when given both the commit message and code diff (RQ1), while few-shot and chain-of-thought prompting further improve performance (RQ2). Embedding-based classifiers achieve the strongest overall results (RQ3).

Applying the approach to 49 open-source projects produced a *Less-Noisy* dataset in which code metrics distinguish Buggy from NotBuggy methods substantially more strongly than in the original *Noisy* dataset (RQ4).

Findings: LLMs can effectively identify tangled method-level changes. Removing these changes improves label quality and makes code metrics substantially more discriminative, providing a practical approach for constructing higher-quality method-level bug datasets.

5.1.2 How Far Are We from Method-Level Bug Prediction?

Chapter 3 examines whether cleaner labels, richer representations, and realistic evaluation are sufficient to make method-level bug prediction reliable. Across 100 open-source Java projects, we combine LLM-based untangling, software metrics, source-code embeddings, and realistic project-disjoint and temporal evaluation.

Cleaner labels improve inception-time test MCC from 0.13 to 0.16. Adding code

embeddings increases validation MCC to 0.26, but the best held-out test result reaches only 0.18. In the Evolution setting, embeddings underperform the tabular baseline under project-disjoint evaluation (0.12 versus 0.15 MCC), while within-project temporal evaluation reaches 0.224 MCC.

Findings: Cleaner labels and richer representations improve prediction, but realistic test performance remains limited. Severe class imbalance and weak cross-project generalization are the main remaining bottlenecks, leaving method-level bug prediction an open problem.

5.1.3 Predicting App Popularity at Launch

Chapter 4 investigates whether internal software properties available at launch can predict future app popularity. We construct a dataset of 446 open-source Java-based Android apps by linking source-code information from F-Droid with ratings and downloads from Google Play.

Regression models fail to predict exact ratings or yearly download counts reliably. However, when popularity is reformulated as a binary classification task, the available features provide moderate predictive signal, reaching F1-scores of 0.72 for rating-based popularity and 0.69 for download-based popularity.

Findings: Internal software metrics cannot reliably predict exact app popularity, but they provide moderate signal for distinguishing Popular from Unpopular apps. They should therefore be treated as one predictor among several rather than as a standalone indicator of market success.

Contribution Summary

Summary: Across the three studies, internal software metrics are neither universally effective nor universally ineffective. Better labels make their relationship with bug-proneness clearer, but realistic method-level bug prediction remains limited by imbalance and poor cross-project generalization. For app popularity, exact prediction fails, while coarser classification remains feasible. These findings show that the value of software metrics depends strongly on the prediction context and how the prediction task is formulated and evaluated.

5.2 Future Work

Our findings identify two main directions for future research: improving generalization in method-level bug prediction and extending app popularity prediction beyond internal software metrics.

5.2.1 A Generalizable Method-Level Bug Prediction Model

Future work should directly address the two main bottlenecks identified in our study: severe class imbalance and weak cross-project generalization.

First, imbalance-aware learning should be explored beyond conventional resampling. Use of generative AI models to synthesize realistic buggy methods is another possible way to improve the representation of the rare buggy class and solve the imbalanced dataset issue.

Second, project-specific adaptation appears promising. Models could be trained across many projects and then adapted using the target project’s own history. The

stronger performance under within-project temporal evaluation suggests that project-specific information may capture development practices and defect patterns that generic cross-project models miss.

Finally, LLMs could be investigated as history-aware predictors. By jointly analyzing source code and its evolution, they may capture semantic and temporal signals that static metrics and embeddings fail to represent.

5.2.2 Improving App Popularity Prediction

Future work should combine internal software metrics with non-code factors that influence popularity, including developer reputation, genre competition, UI quality, pricing, and marketing. The key question is how much predictive value these factors add beyond the source code.

Methodologically, future studies should further investigate feature selection and imbalance handling, both of which affect model performance. Genre-specific models are another promising direction because genre emerged as an important predictor.

The primary limitation is data availability. Enlarging the set of apps available across both F-Droid and Google Play would support more reliable genre-specific models. Extending the analysis to other programming languages, platforms, and software domains would also test the generalizability of our findings.

Bibliography

- [1] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol. An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension. In *2011 15th European Conference on Software Maintenance and Reengineering*, pages 181–190, 2011. ISSN: 1534-5351.
- [2] A. Abdu, Z. Zhai, H. A. Abdo, R. Algabri, M. A. Al-Masni, M. S. Muhammad, and Y. H. Gu. Semantic and traditional feature fusion for software defect prediction using hybrid deep learning model. *Scientific Reports*, 14(1):14771, 2024.
- [3] S. Agarwal, L. Ahmad, J. Ai, S. Altman, A. Applebaum, E. Arbus, R. K. Arora, Y. Bai, B. Baker, H. Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.
- [4] 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*, 220:112263, 2025.
- [5] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. Unified Pre-training for Program Understanding and Generation, 2021. arXiv:2103.06333.

- [6] T. Aladics, J. Jász, and R. Ferenc. Bug prediction using source code embedding based on doc2vec. In *International Conference on Computational Science and Its Applications*, pages 382–397, 2021.
- [7] M. Alenezi and I. Almomani. Empirical analysis of static code metrics for predicting risk scores in android applications. In *5th International Symposium on Data Mining Applications*, pages 84–94, 2018.
- [8] A. Alhejaili and J. Blustein. A Study on How Users Choose Apps. In *Human-Computer Interaction. User Experience and Behavior*, pages 3–22, 2022. ISSN: 1611-3349.
- [9] M. Ali, M. E. Joorabchi, and A. Mesbah. Same app, different app stores: A comparative study. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 79–90, 2017.
- [10] H. Alsolai and M. Roper. A systematic literature review of machine learning techniques for software maintainability prediction. *INFORMATION AND SOFTWARE TECHNOLOGY*, 119:106214, 2020. Num Pages: 25 Place: Amsterdam Publisher: Elsevier Web of Science ID: WOS:000513290100007.
- [11] 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*, pages 716–721, 2018.
- [12] T. L. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark

- data. In *IEEE International Conference on Software Maintenance*, pages 1–10, 2010.
- [13] X. Amatriain. Prompt Design and Engineering: Introduction and Advanced Methods, 2024. arXiv:2401.14423.
- [14] M. Aniche. Java code metrics calculator (CK), 2015. URL <https://github.com/mauricioaniche/ck/>.
- [15] Apple. Ratings and reviews overview - Monitor ratings and reviews - App Store Connect - Help - Apple Developer, Oct. 2024. URL <https://developer.apple.com/help/app-store-connect/monitor-ratings-and-reviews/ratings-and-reviews-overview/>.
- [16] J. Arevalo, T. Solorio, M. Montes-y Gómez, and F. A. González. Gated multi-modal units for information fusion. In *ICLR Workshop*, 2017.
- [17] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, 108:115–138, 2019.
- [18] J. Bae, S. Kwon, and S. Myeong. Enhancing software code vulnerability detection using gpt-4o and claude-3.5 sonnet: A study on prompt engineering techniques. *Electronics*, 13(13):2657, 2024.
- [19] A. A. Bangash, H. Sahar, A. Hindle, and K. Ali. On the time-based conclusion stability of cross-project defect prediction models. *Empirical Software Engineering*, 25(6):5047–5083, 2020.

-
- [20] M. C. Barbieri, B. I. Grisci, and M. Dorn. Analysis and comparison of feature selection methods towards performance and stability. *Expert Systems with Applications*, 249:123667, 2024.
- [21] V. Basili, L. Briand, and W. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.
- [22] 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*, pages 351–361, 2024.
- [23] G. Bavota, M. Linares-Vasquez, C. E. Bernal-Cardenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk. The impact of API change- and fault-proneness on the user ratings of Android apps. *IEEE Transactions on Software Engineering*, 41(4), pp. 384-407, 2015.
- [24] R. M. Bell, T. J. Ostrand, and E. J. Weyuker. Does measuring code change improve fault prediction? In *Proceedings of the 7th international conference on predictive models in software engineering*, pages 1–8, 2011.
- [25] 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)*, 57(1):289–300, 1995.

-
- [26] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *The journal of machine learning research*, 13(1):281–305, 2012.
- [27] Z. Bi, K. Chen, C.-Y. Tseng, D. Zhang, T. Wang, H. Luo, L. Chen, J. Huang, J. Guan, J. Hao, et al. Is gpt-oss good? a comprehensive evaluation of openai’s latest open source models. *arXiv preprint arXiv:2508.12461*, 2025.
- [28] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced? bias in bug-fix datasets. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 121–130, 2009.
- [29] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don’t Touch My Code! Examining the Effects of Ownership on Software Quality. In *ESEC/FSE 2011*, pp. 4-14, 2011.
- [30] L. Breiman. Random Forests. *Machine Learning*, 45(1), pp. 5-32, 2001.
- [31] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. 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. 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 *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901, 2020.

-
- [32] M. Bruni, F. Gabrielli, M. Ghafari, and M. Kropp. Benchmarking Prompt Engineering Techniques for Secure Code Generation with GPT Models, 2025. arXiv:2502.06039.
- [33] R. P. Buse and W. R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010.
- [34] Y. Cai and R. Kazman. Software design analysis and technical debt management based on design rule theory. *Information and software technology*, 164: 107322, 2023.
- [35] Y. Cai and R. Kazman. DV8: Automated Architecture Analysis Tool Suites. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 53–54, 2019.
- [36] D. Camilo, E. Lasso, A. Ledezma, and J. C. Corrales. Feature selection for classification tasks: Expert knowledge or traditional methods? *Journal of Intelligent & Fuzzy Systems*, 34(5):2825–2835, 2018.
- [37] B. Carbunar and R. Potharaju. A longitudinal study of the google app market. In *Proceedings of the 2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2015*, pages 242–249, 2015.
- [38] G. Catolino. Does source code quality reflect the ratings of apps? In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, pages 43–44, Gothenburg Sweden, 2018.

-
- [39] G. Catolino, F. Palomba, F. A. Fontana, A. De Lucia, A. Zaidman, and F. Ferrucci. Improving change prediction models with code smell-related information. *Empirical Software Engineering*, 25(1):49–95, 2020.
- [40] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.
- [41] S. Chen, S. Xu, Y. Yao, and F. Xu. Untangling Composite Commits by Attributed Graph Clustering. In *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, pages 117–126, 2022.
- [42] T. Chen and C. Guestrin. XGBoost: A Scalable Tree Boosting System. In *KDD 2016*, pp. 785–794, 2016.
- [43] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1580–1596, 2020.
- [44] R. Chew, J. Bollenbacher, M. Wenger, J. Speer, and A. Kim. LLM-Assisted Content Analysis: Using Large Language Models to Support Deductive Coding, 2023. arXiv:2306.14924.
- [45] D. Chicco and G. Jurman. The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. *BMC Genomics*, 21, Article 6, 2020.

-
- [46] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
- [47] S. Chowdhury. The Good, the Bad, and the Monstrous: Predicting Highly Change-Prone Source Code Methods at Their Inception. *ACM Trans. Softw. Eng. Methodol.*, 2025.
- [48] S. Chowdhury, S. D. Nardo, A. Hindle, and Z. M. J. Jiang. An exploratory study on assessing the energy impact of logging on Android applications. *EMSE*, 23(3), pp. 1422-1456, 2018.
- [49] S. Chowdhury, S. Borle, S. Romansky, and A. Hindle. Greenscaler: training software energy models with automatic test generation. *Empirical Software Engineering*, 24(4):1649–1692, 2019.
- [50] S. Chowdhury, R. Holmes, A. Zaidman, and R. Kazman. Revisiting the debate: Are code metrics useful for measuring maintenance effort? *Empirical Software Engineering*, 27(6):158, 2022.
- [51] S. Chowdhury, G. Uddin, H. Hemmati, and R. Holmes. Method-level Bug Prediction: Problems and Promises. *ACM Transactions on Software Engineering and Methodology*, 33(4):1–31, 2024.
- [52] 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)*, pages 813–825, 2025.

- [53] 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*, pages 252–264, 2022.
- [54] christine fisher. Boeing found another software bug on the 737 Max, Feb. 2020. URL <https://www.engadget.com/2020-02-06-boeing-737-max-software-bug.html>. [Online; last accessed 2025-07-18].
- [55] J. Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.
- [56] W. J. Conover. *Practical nonparametric statistics*. john wiley & sons, 1999.
- [57] R. Coppola, L. Ardito, and M. Torchiano. Characterizing the transition to kotlin of android apps: a study on f-droid, play store, and github. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics*, pages 8–14, 2019.
- [58] L. Corral and I. Fronza. Better Code for Better Apps: A Study on Source Code Quality and Market Success of Android Applications. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, pages 22–32, Florence, Italy, 2015.
- [59] L. Cruz, R. Abreu, and D. Lo. To the attention of mobile software developers: guess what, test your app! *Empirical Software Engineering*, 24:2438–2468, 2019.

- [60] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love. Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics. *IEEE Transactions on Software Engineering*, SE-5(2): 96–104, 1979.
- [61] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Transactions on Software Engineering*, 43(7), pp. 641–657, 2017.
- [62] M. D’Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *MSR 2010*, pp. 31–41, 2010.
- [63] A. Danial. cloc: 1.92, Dec. 2021. URL <https://doi.org/10.5281/zenodo.5760077>.
- [64] A. Di Sorbo, G. Grano, C. Aaron Visaggio, and S. Panichella. Investigating the criticality of user-reported issues through their relations with app rating. *Journal of Software: Evolution and Process*, 33(3):e2316, 2021.
- [65] M. Dias, A. Bacchelli, G. Gousios, D. Cassou, and S. Ducasse. Untangling fine-grained code changes. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering*, pages 341–350, 2015.
- [66] G. J. Digirolamo and D. L. Hintzman. First impressions are lasting impressions: A primacy effect in memory for repetitions. *Psychonomic Bulletin & Review*, 4(1):121–124, 1997.

-
- [67] K. El Emam, S. Benlarbi, N. Goel, and S. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650, 2001. Conference Name: IEEE Transactions on Software Engineering.
- [68] S. Elmidaoui, L. Cheikhi, A. Idri, and A. Abran. Machine learning techniques for software maintainability prediction: Accuracy analysis. *Journal of Computer Science and Technology*, 35(5):1147–1174, 2020.
- [69] L. Erlikh. Leveraging Legacy System Dollars for E-Business. *IT Professional*, 2(3), pp. 17-23, 2000.
- [70] M. Fan, W. Zhang, H. Zhao, G. Liang, and Z. Jin. Detect Hidden Dependency to Untangle Commits. In *2024 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 179–190, 2024.
- [71] S. Feng, S. Ma, H. Wang, D. Kong, and C. Chen. Mud: Towards a large-scale and noise-filtered ui dataset for modern style ui modeling. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2024.
- [72] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.
- [73] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5), pp. 675-689, 1999.

- [74] R. Ferenc, D. Ban, T. Grosz, and T. Gyimothy. Deep learning in static, metric-based bug prediction. *ARRAY*, 6:100021, 2020. Num Pages: 9 Place: Amsterdam Publisher: Elsevier Web of Science ID: WOS:001140484600003.
- [75] 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*, 169:110691, 2020.
- [76] B. Fu, J. Lin, L. Li, C. Faloutsos, J. Hong, and N. Sadeh. Why people hate your app: Making sense of user feedback in a mobile app store. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1276–1284, 2013.
- [77] J. Gao, P. Li, Z. Chen, and J. Zhang. A survey on deep learning for multimodal data fusion. *Neural computation*, 32(5):829–864, 2020.
- [78] F.-X. Geiger, I. Malavolta, L. Pascarella, F. Palomba, D. Di Nucci, and A. Bacchelli. A graph-based dataset of commit history of real-world android apps. In *Proceedings of the 15th international conference on mining software repositories*, pages 30–33, 2018.
- [79] B. Gezici, N. Bölücü, A. Tarhan, and B. Can. Neural sentiment analysis of user reviews to predict user ratings. In *2019 4th International conference on computer science and engineering (UBMK)*, pages 629–634, 2019.
- [80] B. Gezici, A. Tarhan, and O. Chouseinoglou. Internal and external quality in

- the evolution of mobile software: An exploratory study in open-source market. *Information and software technology*, 112:178–200, 2019.
- [81] 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*, pages 171–180, 2012.
- [82] J. Y. Gil and G. Lalouche. When do Software Complexity Metrics Mean Nothing? – When Examined out of Context. *The Journal of Object Technology*, 15(1):2:1, 2016.
- [83] Y. Gil and G. Lalouche. On the correlation between size and metric validity. *Empirical Software Engineering*, 22:1–27, 2017.
- [84] L. Giray. Prompt Engineering with ChatGPT: A Guide for Academic Writers. *Annals of Biomedical Engineering*, 51(12):2629–2633, 2023.
- [85] Google. Google Play, Oct. 2024. URL <https://play.google/intl/en/comment-posting-policy/>.
- [86] Google DeepMind. Advancing Gemini: December 2024 Update, December 2024. URL <https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/>. [Online; last accessed: 2025-07-11].
- [87] G. Grano, A. Di Sorbo, F. Mercaldo, C. A. Visaggio, G. Canfora, and S. Panichella. Android apps and user feedback: a dataset for software evolution

- and quality improvement. In *Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics*, pages 8–11, Paderborn Germany, 2017.
- [88] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7), pp. 653-661, 2000.
- [89] 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*, pages 1510–1522, 2021.
- [90] J. Gui, S. Mcilroy, M. Nagappan, and W. G. J. Halfond. Truth in Advertising: The Hidden Cost of Mobile Ads for Software Developers. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 100–110, 2015. ISSN: 1558-1225.
- [91] B. Guo and M. Song. Interactively Decomposing Composite Changes to Support Code Review and Regression Testing. In *2017 IEEE 41st Annual Computer Software and Applications Conference*, volume 1, pages 118–127, 2017.
- [92] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.
- [93] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented

- metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10), pp. 897-910, 2005.
- [94] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2012.
- [95] L. Hao, X. Li, Y. Tan, and J. Xu. The economic value of ratings in app market. *Available at SSRN 1892584*, 2011.
- [96] M. Harman, Y. Jia, and Y. Zhang. App store mining and analysis: Msr for app stores. In *2012 9th IEEE working conference on mining software repositories (MSR)*, pages 108–111, 2012.
- [97] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry. The Strengths and Behavioral Quirks of Java Bytecode Decompilers. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 92–102, 2019. ISSN: 2470-6892.
- [98] H. Hata, O. Mizuno, and T. Kikuno. Bug prediction based on fine-grained module histories. In *2012 34th international conference on software engineering (ICSE)*, pages 200–210. IEEE, 2012.
- [99] B. He, H. Xu, L. Jin, G. Guo, Y. Chen, and G. Weng. An Investigation into Android In-App Ad Practice: Implications for App Developers. In *IEEE IN-FOCOM 2018 - IEEE Conference on Computer Communications*, pages 2465–2473, 2018.

- [100] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien. Tracking the software quality of android applications along their evolution (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 236–247. IEEE, 2015.
- [101] S. Herbold, A. Trautsch, B. Ledel, A. Aghamohammadi, T. A. Ghaleb, K. K. Chahal, T. Bossenmaier, B. Nagaria, P. Makedonski, M. N. Ahmadabadi, K. Szabados, H. Spieker, M. Madeja, N. Hoy, V. Lenarduzzi, S. Wang, G. Rodríguez-Pérez, R. Colomo-Palacios, R. Verdecchia, P. Singh, Y. Qin, D. Chakroborti, W. Davis, V. Walunj, H. Wu, D. Marcilio, O. Alam, A. Aldaej, I. Amit, B. Turhan, S. Eismann, A.-K. Wickert, I. Malavolta, M. Sulír, F. Fard, A. Z. Henley, S. Kourtzanidis, E. Tuzun, C. Treude, S. M. Shamasbi, I. Pashchenko, M. Wyrich, J. Davis, A. Serebrenik, E. Albrecht, E. U. Aktas, D. Strüber, and J. Erbel. A fine-grained data set and analysis of tangling in bug fixing commits. *Empirical Software Engineering*, 27(6):125, 2022.
- [102] I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles. Towards a Theoretical Model for Software Growth. In *Fourth International Workshop on Mining Software Repositories (MSR’07:ICSE Workshops 2007)*, pages 21–21, 2007. ISSN: 2160-1860.
- [103] K. Herzig and A. Zeller. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories*, pages 121–130, 2013.
- [104] K. Herzig, S. Just, and A. Zeller. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering*, 21(2):303–336, 2016.

-
- [105] M. R. Hess and J. D. Kromrey. Robust confidence intervals for effect sizes: A comparative study of cohen'sd and cliff's delta under non-normality and heterogeneous variances. In *annual meeting of the American Educational Research Association*, volume 1, 2004.
- [106] T. Hu and N. Collier. Quantifying the Persona Effect in LLM Simulations, 2024. arXiv:2402.10811.
- [107] L. J. Human, G. M. Sandstrom, J. C. Biesanz, and E. W. Dunn. Accurate First Impressions Leave a Lasting Impression: The Long-Term Effects of Distinctive Self-Other Agreement on Relationship Development. *Social Psychological and Personality Science*, 4(4):395–402, 2013. Publisher: SAGE Publications Inc.
- [108] X. Huo, Y. Yang, M. Li, and D.-C. Zhan. Learning semantic features for software defect prediction by code comments embedding. In *2018 IEEE ICDM*, pages 1049–1054. IEEE, 2018.
- [109] S. Ickin, K. Petersen, and J. Gonzalez-Huerta. Why do users install and delete apps? a survey study. In *Software Business: 8th International Conference, ICSOB 2017, Essen, Germany, June 12-13, 2017, Proceedings 8*, pages 186–191. Springer, 2017.
- [110] U. Iftikhar, N. B. Ali, J. Börstler, and M. Usman. A tertiary study on links between source code metrics and external quality attributes. *Information and Software Technology*, 165:107348, 2024.
- [111] 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 *ICSE 2026*, 2026.
- [112] I. Jaoua, O. B. Sghaier, and H. Sahraoui. Combining large language models with static analyzers for code review generation. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories*, pages 174–186, 2025.
- [113] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.*, 55(12):248:1–248:38, 2023.
- [114] M. Jimenez, R. Rwemalika, M. Papadakis, F. Sarro, Y. L. Traon, and M. Harman. The Importance of Accounting for Real-World Labelling When Predicting Software Vulnerabilities. In *ESEC/FSE 2019*, pp. 695-705, 2019.
- [115] R. C. Jisha, R. Krishnan, and V. Vikraman. Mobile Applications Recommendation Based on User Ratings and Permissions. In *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 1000–1005, 2018.
- [116] D. Kafura and G. R. Reddy. The use of software complexity metrics in software maintenance. *IEEE Transactions on Software Engineering*, SE-13(3), pp. 335-343, 1987.
- [117] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The Promises and Perils of Mining GitHub. In *MSR 2014*, pp. 92-101, 2014.

-
- [118] Y. Kamei and E. Shihab. Defect Prediction: Accomplishments and Future Challenges. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, volume 5, pages 33–45, 2016.
- [119] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *IEEE Transactions on Software Engineering*, 39(6), pp. 757–773, 2013.
- [120] M. Karagkiozidou, C. Ziakis, M. Vlachopoulou, and T. Kyrkoudis. App Store Optimization Factors for Effective Mobile App Ranking. In A. Kavoura, E. Kefallonitis, and A. Giovanis, editors, *Strategic Innovative Marketing and Tourism*, pages 479–486, Cham, 2019.
- [121] I. Keraghel, S. Morbieu, and M. Nadif. Beyond Words: A Comparative Analysis of LLM Embeddings for Effective Clustering. In *Advances in Intelligent Data Analysis XXII*, pages 205–216, 2024.
- [122] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan. What Do Mobile App Users Complain About? *IEEE Software*, 32(3):70–77, 2015. Conference Name: IEEE Software.
- [123] 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*, pages 75–84, 2009.
- [124] 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*, 17(3):243–275, 2012.
- [125] S. Kim, T. Zimmermann, K. Pan, and E. J. W. Jr. Automatic identification of bug-introducing changes. In *ASE 2006, pp. 81-90*, 2006.
- [126] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto. Hey! are you committing tangled changes? In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 262–265, 2014.
- [127] P. S. Kochhar, Y. Tian, and D. Lo. Potential biases in bug localization: Do they matter? In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 803–814, 2014.
- [128] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.
- [129] S. K. Kuttal, Y. Bai, E. Scott, and R. Sharma. Tug of perspectives: Mobile app users vs developers. *International Journal of Computer Science and Information Security (IJCSIS)*, 18(6), 2020.
- [130] D. Landman, A. Serebrenik, and J. Vinju. Empirical Analysis of the Relationship between CC and SLOC in a Large Corpus of Java Methods. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 221–230, 2014. ISSN: 1063-6773.
- [131] M. Lanza, A. Mocci, and L. Ponzanelli. The tragedy of defect prediction, prince

- of empirical software engineering research. *IEEE Software*, 33(6), pp. 102-105, 2016.
- [132] J. Lee, Z. Dai, X. Ren, B. Chen, D. Cer, J. R. Cole, K. Hui, M. Boratko, R. Kapadia, W. Ding, et al. Gecko: Versatile text embeddings distilled from large language models. *arXiv preprint arXiv:2403.20327*, 2024.
- [133] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *Proceedings of the 4th International Symposium on Software Metrics, METRICS '97*, page 20, USA, 1997.
- [134] G. Lemaître, F. Nogueira, and C. K. Aridas. Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning. *Journal of Machine Learning Research*, 18(17), pp. 1-5, 2017.
- [135] M. Levy, A. Jacoby, and Y. Goldberg. Same Task, More Tokens: The Impact of Input Length on the Reasoning Performance of Large Language Models, 2024. arXiv:2402.14848.
- [136] W. Li and S. Henry. Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2):111–122, 1993.
- [137] Y. Li, T. Li, and H. Liu. Recent advances in feature selection and its applications. *Knowledge and Information Systems*, 53(3):551–577, 2017.
- [138] Y. Li, S. Wang, and T. N. Nguyen. UTANGO: Untangling commits with context-aware, graph-based, code change clustering learning model. In *Pro-*

- ceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 221–232, 2022.
- [139] Y. Li, Q. Luo, P. Wu, and H. Zheng. Vdmaf: Cross-language source code vulnerability detection using multi-head attention fusion. *Information and Software Technology*, 183:107739, 2025.
- [140] T. Liang, L. Chen, X. Ying, P. S. Yu, J. Wu, and Z. Zheng. Mobile Application Rating Prediction via Feature-Oriented Matrix Factorization. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 261–268, 2017.
- [141] E. Liptrot, H. A. Pearson, A. Montazami, and A. K. Dubé. Why this app? How user ratings and app store rankings impact educators’ selection of educational apps. *Computers & Education*, 218:105080, 2024.
- [142] A. Liu, J. Lefever, Y. Han, and Y. Cai. Prevalence and severity of design anti-patterns in open source programs-A large-scale study. *INFORMATION AND SOFTWARE TECHNOLOGY*, 170:107429, 2024. Num Pages: 15 Place: Amsterdam Publisher: Elsevier Web of Science ID: WOS:001222153300001.
- [143] 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 Comput. Surv.*, 55(9):195:1–195:35, 2023.
- [144] X. Liu, L. Huang, C. Li, and V. Ng. Linking Source Code to Untangled Change Intents. In *2018 IEEE International Conference on Software Maintenance and Evolution*, pages 393–403, 2018.

-
- [145] Y. Liu, D. Li, K. Wang, Z. Xiong, F. Shi, J. Wang, B. Li, and B. Hang. Are LLMs good at structured outputs? A benchmark for evaluating structured output capabilities in LLMs. *Information Processing & Management*, 61(5):103809, 2024.
- [146] W. Ma, S. Liu, M. Zhao, X. Xie, W. Wang, Q. Hu, J. Zhang, and Y. Liu. Unveiling code pre-trained models: Investigating syntax and semantics capacities. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–29, 2024.
- [147] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma. Cliff’s delta calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica*, 10(2):545–555, 2011.
- [148] A. MacCormack, J. Rusnak, and C. Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015–1030, 2006.
- [149] A. Mahmood. Identifying the influence of various factor of apps on google play apps ratings. *Journal of Data, Information and Management*, 2(1):15–23, 2020.
- [150] E. Mashhadi, H. Ahmadvand, and H. Hemmati. Method-level bug severity prediction using source code metrics and llms. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 635–646, 2023.
- [151] 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*, 217:112179, 2024.
- [152] B. W. Matthews. Comparison of the predicted and observed secondary structure of T4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405(2), pp. 442-451, 1975.
- [153] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, pages 308–320, 1976.
- [154] P. E. McKnight and J. Najab. Mann-whitney u test. *The Corsini encyclopedia of psychology*, pages 1–1, 2010.
- [155] Q. McNemar. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157, 1947.
- [156] S. Mehta, L. Kumar, S. Misra, K. S. Patnaik, and V. Singh. Nested deep learning with learned network embeddings for software defect prediction. *Applied Soft Computing*, 174:113057, 2025.
- [157] H. Mende, M. Frye, P.-A. Vogel, S. Kiroriwal, R. H. Schmitt, and T. Bergs. On the importance of domain expertise in feature engineering for predictive product quality in production. *Procedia CIRP*, 118:1096–1101, 2023.
- [158] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1): 2–13, 2007.

-
- [159] J. K. Miller, D. L. Westerman, and M. E. Lloyd. Are first impressions lasting impressions? An exploration of the generality of the primacy effect in memory for repetitions. *Memory & Cognition*, 32(8):1305–1315, 2004.
- [160] 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*, pages 499–510, 2016.
- [161] R. Mo, S. Wei, Q. Feng, and Z. Li. An exploratory study of bug prediction at the method level. *Information and Software Technology*, 144:106794, 2022.
- [162] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2010.
- [163] R. Moser, W. Pedrycz, and G. Succi. Analysis of the reliability of a subset of change metrics for defect prediction. In *ESEM 2008*, pp. 309–311, 2008.
- [164] W. Muylaert and C. De Roover. [Research Paper] Untangling Composite Commits Using Program Slicing. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation*, pages 193–202, 2018.
- [165] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *ICSE 2005*, pp. 284–292, 2005.
- [166] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *ICSE 2006*, pp. 452–461, 2006.

-
- [167] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers. Using an LLM to Help With Code Understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.
- [168] M. Nayebe, H. Cho, and G. Ruhe. App store mining is not enough for app improvement. *Empirical Software Engineering*, 23:2764–2794, 2018.
- [169] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, and D. Lo. The best of both worlds: integrating semantic features with expert features for defect prediction and localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 672–683, 2022.
- [170] E. Noei, M. D. Syer, Y. Zou, A. E. Hassan, and I. Keivanloo. A study of the relation of mobile device attributes with the user-perceived quality of Android apps. *Empirical Software Engineering*, 22(6):3088–3116, 2017.
- [171] D. D. Nucci, F. Palomba, G. D. Rosa, G. Bavota, R. Oliveto, and A. D. Lucia. A developer centered bug prediction model. *IEEE Transactions on Software Engineering*, 44(1), pp. 5-24, 2018.
- [172] H. M. Olague, L. H. Etzkorn, S. Gholston, and S. Quattlebaum. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Transactions on software Engineering*, 33(6):402–419, 2007.
- [173] P. Oman and J. Hagemeister. Metrics for assessing a software system’s main-

- tainability. In *Proceedings Conference on Software Maintenance 1992*, pages 337–338, 1992.
- [174] OpenAI. GPT-4o mini: Advancing Cost-Efficient Intelligence, July 2025. URL <https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/>. [Online; last accessed: 2025-07-11].
- [175] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto. Smells Like Teen Spirit: Improving Bug Prediction Performance Using the Intensity of Code Smells. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 244–255, 2016.
- [176] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia. Lightweight detection of Android-specific code smells: The aDoctor project. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 487–491, Klagenfurt, Austria, 2017.
- [177] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto. Toward a smell-aware bug prediction model. *IEEE Transactions on Software Engineering*, 45(2):194–218, 2017.
- [178] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, 23(3):1188–1221, 2018.

-
- [179] P.-P. Pârțachi, S. K. Dash, M. Allamanis, and E. T. Barr. Flexeme: Untangling commits using lexical flows. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 63–74, 2020.
- [180] L. Pascarella, F. Palomba, and A. Bacchelli. Re-evaluating Method-Level Bug Prediction. In *SANER 2018*, pp. 592-601, 2018.
- [181] L. Pascarella, F. Palomba, and A. Bacchelli. Fine-grained just-in-time defect prediction. *Journal of Systems and Software*, 150, pp. 22-36, 2019.
- [182] L. Pascarella, F. Palomba, and A. Bacchelli. On the performance of method-level bug prediction: A negative result. *Journal of Systems and Software*, 161: 110493, 2020.
- [183] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.
- [184] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [185] E. Perez, F. Strub, H. de Vries, V. Dumoulin, and A. Courville. FiLM: Vi-

- sual reasoning with a general conditioning layer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.
- [186] A. Petukhova, J. P. Matos-Carvalho, and N. Fachada. Text Clustering with Large Language Model Embeddings, 2024.
- [187] C. Pornprasit and C. Tantithamthavorn. JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction. In *MSR 2021*, pp. 369-379, 2021.
- [188] Y. Qu and H. Yin. Evaluating network embedding techniques’ performances in software bug prediction. *Empirical Software Engineering*, 26(4):60, 2021.
- [189] P. Rachow and M. Riebisch. An architecture smell knowledge base for managing architecture technical debt. In *Proceedings of the International Conference on Technical Debt*, TechDebt ’22, pages 1–10, New York, NY, USA, 2022.
- [190] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397–1418, 2013.
- [191] A. Rahman, P. Pradhan, A. Partho, and L. Williams. Predicting android application security and privacy risk with static code metrics. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 149–153, 2017.
- [192] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *ICSE 2013*, pp. 432-441, 2013.

-
- [193] 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*, pages 131–140, 2017.
- [194] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu. On the “naturalness” of buggy code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 428–439, 2016.
- [195] D. Romano and M. Pinzger. Using source code metrics to predict change-prone Java interfaces. In *ICSM 2011, pp. 303-312*, 2011.
- [196] I. J. M. Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. E. Hassan. Examining the rating system used in mobile-app stores. *Ieee Software*, 33(6): 86–92, 2015.
- [197] J. Sallou, T. Durieux, and A. Panichella. Breaking the Silence: The Threats of Using LLMs in Software Engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 102–106, 2024.
- [198] F. Sarro, M. Harman, Y. Jia, and Y. Zhang. Customer Rating Reactions Can Be Predicted Purely using App Features. In *2018 IEEE 26th International Requirements Engineering Conference (RE)*, pages 76–87, Banff, AB, 2018.
- [199] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto. Improving

- code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, 2016.
- [200] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto. Automatically assessing code understandability: How far are we? In *32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 417–427, 2017.
- [201] M. Scott. Report: Software Failure Caused \$1.7 Trillion in Financial Losses in 2017. Web resource (techrepublic.com), 2018. URL <https://www.techrepublic.com/article/report-software-failure-caused-1-7-trillion-in-financial-losses-in-2017/>.
- [202] F. Servant and J. A. Jones. Fuzzy Fine-Grained Code-History Analysis. In *2017 IEEE/ACM 39th International Conference on Software Engineering*, pages 746–757, 2017.
- [203] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant’Anna. From retrospect to prospect: Assessing modularity and stability from software architecture. In *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, pages 269–272, 2009.
- [204] O. Shaikh, H. Zhang, W. Held, M. Bernstein, and D. Yang. On Second Thought, Let’s Not Think Step by Step! Bias and Toxicity in Zero-Shot Reasoning, 2023. arXiv:2212.08061.
- [205] T. Sharma, P. Mishra, and R. Tiwari. Designite: a software design quality

- assessment tool. In *Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*, pages 1–4, Austin Texas, 2016.
- [206] R. Shatnawi and W. Li. The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. *Journal of systems and software*, 81(11):1868–1882, 2008.
- [207] B. Shen, W. Zhang, C. Kästner, H. Zhao, Z. Wei, G. Liang, and Z. Jin. Smart-Commit: A graph-based interactive assistant for activity-oriented commits. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 379–390, 2021.
- [208] M. Shepperd. A critique of cyclomatic complexity as a software metric. *Softw. Eng. J.*, 3(2):30–36, 1988.
- [209] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6), pp. 772-787, 2011.
- [210] 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*, pages 1–6, 2016.
- [211] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim. Reducing features to

- improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4), pp. 552-569, 2012.
- [212] Y. Singh, A. Kaur, and R. Malhotra. Empirical validation of object-oriented metrics for predicting fault proneness models. *Software quality journal*, 18:3–35, 2010.
- [213] M. Sintaha, N. Nashid, and A. Mesbah. Katana: Dual slicing based context for learning bug fixes. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–27, 2023.
- [214] sixsentix. Most expensive software bugs in history: Sixsentix, 2024. URL <https://www.sixsentix.com/insights/ten-most-expensive-bugs-in-history-part-1>. [Online; last accessed 2025-07-18].
- [215] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *MSR 2005 / ACM SIGSOFT Software Engineering Notes* 30(4), pp. 1-5, 2005.
- [216] SonarQube. SonarQube Overview - OCT 2023, 2023. URL <https://sonarsource.navattic.com/kdq70t3x>.
- [217] 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)*, pages 1–12, 2018. ISSN: 2576-3148.

-
- [218] D. Steidl, B. Hummel, and E. Juergens. Incremental origin analysis of source code files. In *MSR 2014*, pp. 42-51, 2014.
- [219] S. Sun, Y. Li, L. Chen, Y. Zhou, and J. Zhao. Boosting Code-line-level Defect Prediction with Spectrum Information and Causality Analysis. In *2025 IEEE/ACM 47th International Conference on Software Engineering*, pages 776–776, 2025.
- [220] M. D. Syer, M. Nagappan, B. Adams, and A. E. Hassan. Studying the relationship between source code quality and mobile platform dependence. *Software Quality Journal*, 23(3):485–508, 2015.
- [221] Y. Tao and S. Kim. Partitioning Composite Code Changes to Facilitate Code Review. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 180–190, 2015.
- [222] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan. What are the characteristics of high-rated apps? a case study on free android applications. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pages 301–310, 2015.
- [223] U. Tiwari and S. Kumar. Cyclomatic complexity metric for component based software. *SIGSOFT Softw. Eng. Notes*, 39(1):1–6, 2014.
- [224] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural ma-

- chine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29, 2019.
- [225] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. ukasz Kaiser, and I. Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.
- [226] H. S. Vera, S. Dua, B. Zhang, D. Salz, R. Mullins, S. R. Panyam, S. Smoot, I. Naim, J. Zou, F. Chen, et al. Embeddinggemma: Powerful and lightweight text representations. *arXiv preprint arXiv:2509.20354*, 2025.
- [227] N. Viennot, E. Garcia, and J. Nieh. A measurement study of google play. In *The 2014 ACM international conference on Measurement and modeling of computer systems*, pages 221–233, 2014.
- [228] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, and X. Yang. Perceptions, expectations, and challenges in defect prediction. *IEEE Transactions on Software Engineering*, 46(11), pp. 1241-1266, 2020.
- [229] H. Wang, Z. Gao, X. Hu, D. Lo, J. Grundy, and X. Wang. Just-in-time todo-missed commits detection. *IEEE Transactions on Software Engineering*, 50(11): 2732–2752, 2024.
- [230] R. Wang, Z. Wang, B. Tang, L. Zhao, and L. Wang. SmartPI: Understanding Permission Implications of Android Apps from User Reviews. *IEEE Transactions on Mobile Computing*, 19(12):2933–2945, 2020.

-
- [231] S. Wang, T. Liu, and L. Tan. Automatically Learning Semantic Features for Defect Prediction. In *ICSE 2016*, pp. 297-308, 2016.
- [232] S. Wang, J. Wang, J. Nam, and N. Nagappan. Continuous software bug prediction. In *ESEM 2021, Article 14*, 2021.
- [233] 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*, 9(4):3835–3855, 2023.
- [234] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto. Predicting Defective Lines Using a Model-Agnostic Technique. *IEEE Transactions on Software Engineering*, 48(5):1480–1496, 2022.
- [235] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le. Finetuned Language Models Are Zero-Shot Learners, 2022. arXiv:2109.01652.
- [236] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models, 2023. arXiv:2201.11903.
- [237] S.-B. Xu, S.-Y. Chen, Y. Yao, and F. Xu. Detecting and Untangling Composite Commits via Attributed Graph Modeling. *Journal of Computer Science and Technology*, 40(1):119–137, 2025.
- [238] A. Yamashita and L. Moonen. Exploring the impact of inter-smell relations

- on software maintainability: An empirical study. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 682–691, 2013. ISSN: 1558-1225.
- [239] J.-Y. Yao, K.-P. Ning, Z.-H. Liu, M.-N. Ning, Y.-Y. Liu, and L. Yuan. LLM Lies: Hallucinations are not Bugs, but Features as Adversarial Examples, 2024. arXiv:2310.01469.
- [240] H. Ye, T. Liu, A. Zhang, W. Hua, and W. Jia. Cognitive Mirage: A Review of Hallucinations in Large Language Models, 2023. arXiv:2309.06794.
- [241] S. Yin, S. Guo, H. Li, C. Li, R. Chen, X. Li, and H. Jiang. Line-Level Defect Prediction by Capturing Code Contexts With Graph Convolutional Networks. *IEEE Transactions on Software Engineering*, 51(1):172–191, 2025.
- [242] G. You, G. Kim, S.-j. Cho, and H. Han. A Comparative Study on Optimization, Obfuscation, and Deobfuscation Tools in Android. *Journal of Internet Services and Information Security*, 11(1):2–15, 2021.
- [243] Q. Zeng, Y. Zhang, Z. Sun, Y. Guo, and H. Liu. Colare: Commit classification via fine-grained context-aware representation of code changes. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 752–763. IEEE, 2024.
- [244] Q. Zeng, Y. Zhang, Z. Qiu, and H. Liu. A first look at conventional commits classification. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, pages 2277–2289, 2025.

-
- [245] Y. Zeng, J. Chen, W. Shang, and T.-H. Chen. Studying the characteristics of logging practices in mobile apps: a case study on f-droid. *Empirical Software Engineering*, 24:3394–3434, 2019.
- [246] 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*, pages 350–359, 2013.
- [247] X. Zhang, Y. Zhou, and C. Zhu. An empirical study of the impact of bad designs on defect proneness. In *2017 International conference on software analysis, testing and evolution (SATE)*, pages 1–9, 2017.
- [248] X. Zhang, Y. Chen, S. Hu, Z. Xu, J. Chen, M. Hao, X. Han, Z. Thai, S. Wang, Z. Liu, et al. ∞ bench: Extending long context evaluation beyond 100k tokens. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15262–15277, 2024.
- [249] Y. Zhang, C. Ge, H. Liu, and K. Zheng. Code smell detection based on supervised learning models: A survey. *Neurocomputing*, 565:127014, 2024.
- [250] Y. Zhang, M. Li, D. Long, X. Zhang, H. Lin, B. Yang, P. Xie, A. Yang, D. Liu, J. Lin, F. Huang, and J. Zhou. Qwen3 Embedding: Advancing Text Embedding and Reranking Through Foundation Models. *arXiv preprint arXiv:2506.05176*, 2025.
- [251] Y. Zhao, L. Gong, Y. Yu, Z. Huang, and M. Wei. An empirical study of

- best practices for code pre-trained models on software engineering classification tasks. *Expert Systems with Applications*, 272:126762, 2025.
- [252] A. Zheng and A. Casari. *Feature engineering for machine learning: principles and techniques for data scientists*. “O’Reilly Media, Inc.”, 2018.
- [253] T. Zheng, H. Liu, H. Xu, X. Chen, P. Yi, and Y. Wu. Few-vuld: A few-shot learning framework for software vulnerability detection. *Computers & Security*, 144:103992, 2024.
- [254] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, C. Cui, O. Bousquet, Q. Le, et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.
- [255] 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*, 83:660–674, 2010.
- [256] W. Zhu, S. Proksch, D. M. German, M. W. Godfrey, L. Li, and S. McIntosh. What is an app store? The software engineering perspective. *Empirical Software Engineering*, 29(1):35, 2024.
- [257] T. Zimmermann, R. Premraj, and A. Zeller. Predicting Defects for Eclipse. In *Third International Workshop on Predictor Models in Software Engineering*, pages 9–9, 2007.