Journal on Information Technologies & Communication **DOI: 10.31276/ICT.2026.0048**

A preliminary study on feature-based test smell detection in Java unit tests

Thi Van Nguyen*

Phenikaa School of Computing, Phenikaa University,

Nguyen Trac Street, Duong Noi Ward, Hanoi, Vietnam

Received 19 June 2026; revised 29 August 2026; accepted 6 September 2026

Abstract:

Maintaining test suites is difficult when test code contains recurring design symptoms such as assertion-heavy tests, tests that exercise several behaviours, hidden dependencies on external resources, or duplicated test logic. This preliminary study examines whether such symptoms can be approximated using a lightweight static-analysis and machine-learning pipeline for Java/JUnit test code. The proposed workflow parses JUnit test methods, derives rule-based candidate labels for four test smells, checks a stratified validation subset, extracts smell-oriented static features, and trains one-vs-rest classifiers based on Logistic Regression, Linear SVM, Random Forest, XGBoost, and stacking. We evaluate the workflow on three Apache Java libraries containing 9,721 JUnit test methods. The preliminary results are interpreted descriptively rather than as evidence of statistical superiority. Stacking is competitive in the within-project setting, whereas XGBoost gives the strongest observed leave-one-project-out performance among the learned models. The findings suggest that feature-based learning is a feasible screening mechanism for JUnit test smell detection, while also showing that label construction and cross-project transfer require cautious interpretation.

Keywords: **JUnit, leave-one-project-out evaluation, machine learning, software quality, static features, test smell detection.**

1. Introduction

Unit tests are a central mechanism for regression testing, continuous integration, and software maintenance. However, test code is also subject to quality degradation. A test may contain many assertions without informative messages, exercise several behaviours in a single method, depend on external resources that are not evident from the test body, or repeat the intent and structure of other tests. These recurring symptoms are generally referred to as

*Email: van.nguyenthi1@phenikaa-uni.edu.vn

test smells. Prior empirical work indicates that test smells can influence maintainability, developers' perception of test quality, and the evolution of test suites [1, 2, 3].

Many available detectors operationalise test smells through manually designed rules. This strategy is practical and easy to deploy, but it is also sensitive to thresholds, coding conventions, and the interpretation of each smell. As a result, outputs from rule-based tools should not be automatically treated as reliable ground truth [4, 5]. At the other end of the spectrum, recent work on language models and code understanding offers promising alternatives, but such approaches often introduce heavier computational requirements and additional replication concerns [6]. For a preliminary empirical setting, a simpler feature-driven pipeline can provide a useful middle ground between rigid rule-based tools and heavier code-understanding models.

This paper investigates a lightweight approach for detecting four JUnit test smells: Assertion Roulette, Eager Test, Mystery Guest, and Duplicate Test. The study treats each smell as a separate binary classification problem at the test-method level. The proposed pipeline extracts test methods from Java projects, constructs candidate labels using smell-specific rules, checks a curated validation subset, computes static features tailored to the target smells, and compares rule-based, traditional machine-learning, boosting, and stacking models under within-project and leave-one-project-out settings. The paper makes the following contributions:

- A reproducible pipeline for extracting and representing JUnit test methods as static feature vectors;

- A weak-label construction procedure with a curated validation subset for four widely studied test smells;

- A comparison of lightweight learning models, including Logistic Regression, Linear SVM, Random Forest, XGBoost, and stacking;

- A preliminary evaluation on three Apache Java libraries, including leave-one-project-out results and feature-group ablation.

The remainder of the paper is organised as follows. Section 2 reviews related work. Section 3 describes the proposed pipeline. Section 4 presents the experimental setup. Section 5 discusses the results and discussion. Section 6 concludes the paper.

2. Related work

Research on test smells has developed along three complementary lines: defining and cataloguing smells, building rule-based detectors, and assessing the reliability and consequences of detected smells. Mapping studies show that Java/JUnit has been the

dominant environment for test smell analysis, partly because the ecosystem provides stable testing conventions and a long history of detection tools [2]. Work relating test-driven development and test smells also remains largely centred on Java/JUnit projects [5]. Recent studies have broadened the discussion to Python and Pytest [7, 8]. These studies motivate a reproducible Java/JUnit setting while also reminding us that conclusions drawn from one ecosystem should be treated cautiously.

A second line of work questions the reliability of smell detectors. A. Panichella, et al. (2022) [3] examined detectability, validity, and reliability twenty years after the original test smell definitions and showed that detector outputs can be inconsistent. Perception studies further suggest that not all smells are equally severe from a developer's viewpoint [4]. H. Damasceno, et al. (2022) [9] analysed test smell refactoring from the developers' perspective and showed that practical refactoring decisions are shaped by context and cost. D. Campos, et al. (2023) [1] also examined test smell survivability and showed that smells may persist across project evolution. Manual-test smells and system-user-interactive test smells have also been discussed in recent work, extending the notion of test quality beyond conventional unit-test code [8, 10]. Overall, these studies suggest that automatic outputs should be treated as decision support rather than unquestionable ground truth.

Data-driven detection has recently become more visible. V. Pontillo, et al. (2024) [11] proposed machine-learning-based test smell detection and reported that learned models can be effective, while cross-project prediction remains difficult. Their results indicate that learned detectors may capture project-specific patterns and that transfer to unseen projects is not guaranteed. Our study follows the same broad direction but focuses on a compact static feature representation and a lightweight pipeline that can be reproduced without heavyweight infrastructure. The goal is not to claim a new learning algorithm, but to assess whether smell-oriented static features are useful for preliminary test-smell screening.

Large language models and code-understanding models have also been explored for smell detection and related code-quality tasks. R. Melo, et al. (2025) [12] investigated agentic language models for test smell hunting, while S. Souza, et al. (2026) [13] examined whether LLMs can move beyond strict rules in code smell detection. B. Zhang, et al. (2024) [6] evaluated parameter-efficient fine-tuning for code smell detection. Work on code smell detection also shows that hyperparameter choices and metric correlations can affect conclusions [14]. These studies are important, but they also raise questions about computational cost, prompt or training stability, and benchmark construction. In contrast, the present work uses transparent static features as a lightweight reference point rather than as a replacement for LLM-based or transformer-based detectors.

Related studies further emphasise the practical effects of poor test quality. I. Nakamura, et al. (2025) [15] investigated self-admitted technical debt in test code. H.K.V. Tran, et al. (2025) [16] examined practitioners' views on quality attributes of test cases and test suites. M.A. Hasan, et al. (2025) [17] studied the prioritisation of test smells using quality metrics and developer perceptions. These references have been checked against the bibliography to avoid citation-reference mismatches. Together, the studies suggest that detection is useful only when the output supports maintainability and prioritisation. Therefore, our approach should be viewed as a screening aid rather than a replacement for human judgement.

3. Proposed methods

3.1. Overview

Figure 1 gives an overview of the proposed workflow. The input consists of Java/JUnit test code, and the unit of analysis is an individual test method. The workflow first extracts test methods and static code signals, then creates candidate labels, derives smell-oriented features, trains one-vs-rest classifiers, and evaluates the models under within-project and cross-project settings.

3.2. Test method extraction

Candidate test files are identified using standard Maven/Gradle test directories, naming conventions, and JUnit imports or annotations. We use JavaParser to extract methods annotated with JUnit 4/5 test annotations, including @Test, @ParameterizedTest, and @RepeatedTest. Dynamic tests based on @TestFactory are excluded because their executable bodies are not always explicit in the source code. The extractor preserves the method body, class context, annotations, assertion calls, method calls, resource-related signals, and a normalised token sequence for similarity analysis.

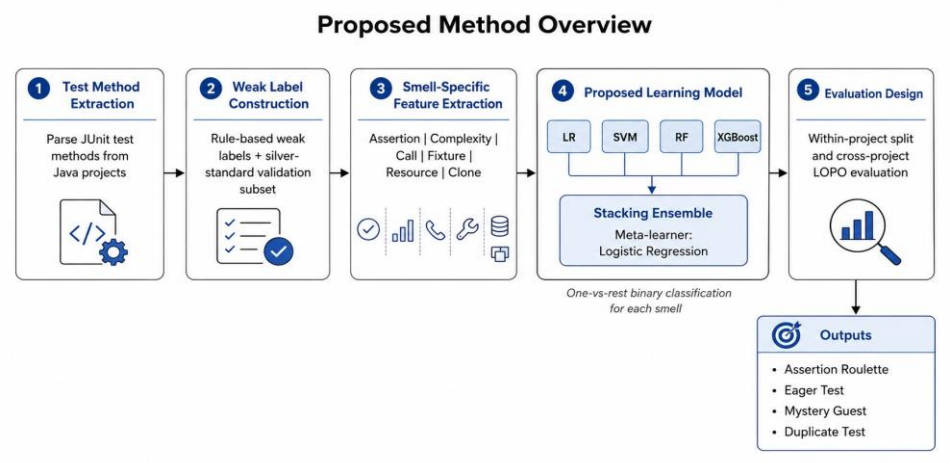


Fig. 1. Overview of the lightweight feature-based pipeline for JUnit test smell detection. Source: Authors.

3.3. Weak label construction

Because fully annotated test smell datasets are costly to construct, we generate weak labels using conservative smell-specific rules and then validate a stratified subset. Assertion Roulette is approximated using assertion counts and missing diagnostic messages. Eager Test is approximated using call-related signals, including the number of approximate production calls and the diversity of called methods. Mystery Guest is approximated through file, resource, database, network, environment, and time-dependency indicators. Duplicate Test is approximated using normalised-token similarity, while very short normalised methods are handled cautiously to reduce false duplicates. These labels are treated as candidate labels, not as complete human ground truth.

3.4. Smell-specific feature extraction

Each test method is encoded as a static feature vector. Assertion features describe assertion counts, diagnostic-message indicators, fluent assertion descriptions, and assertion density. Size and complexity features include lines of code, token counts, branches, loops, object creation, and an approximate cyclomatic measure. Call features summarise total calls, assertion calls, mocking calls, helper calls, distinct called methods, and approximate production calls. Fixture and resource features capture observable setup-related and external-dependency signals. Clone features are computed from normalised token sequences using exact clusters and TF-IDF cosine similarity. Log-transformed and project-normalised variants are added to reduce skew and partially account for project-specific coding style.

3.5. Proposed learning model

Each smell is modelled as a binary one-vs-rest task using the extracted test-method dataset described in section 4.1. We compare a direct rule-based baseline with Logistic Regression, Linear SVM, Random Forest, XGBoost, and a stacking ensemble. The stacking model combines the prediction probabilities of the base learners and uses Logistic Regression as the meta-learner. Linear models use standardised numerical features, whereas tree-based models use the original feature scale. Class imbalance is handled with class weights or the corresponding XGBoost positive-class weighting.

3.6. Evaluation design

We evaluate models in two settings. The within-project setting uses stratified train/test splits inside each project to estimate performance when project-specific coding conventions are shared. The cross-project setting uses leave-one-project-out evaluation, where one project is held out for testing and the remaining projects are used for training. This setting is more demanding and better reflects the use of a detector on a previously unseen project. We

report F1-score and Macro-F1 as primary metrics and MCC as a complementary imbalance-aware measure.

4. Experimental setup

4.1. Dataset

The preliminary dataset contains three Apache Java libraries: Apache Commons Lang, Apache Commons IO, and Apache Commons Collections. All projects are Maven-based and use JUnit 5 annotations in the extracted tests. Each repository was fixed to a specific snapshot before extraction; repository URLs are shown in Table 1, while commit metadata is retained with the extraction logs. A JUnit sample repository used during parser testing was excluded because it is a demonstration project rather than a real-world library.

Table 1. Dataset statistics for the three Apache Java projects used in the study.

Project	Repository	Files	Methods	Param.	Disabled
Commons Lang	apache/commons-lang	296	4,992	139	8
Commons IO	apache/commons-io	244	2,779	236	9
Commons Collections	apache/commons-collections	224	1,950	18	2
Total	–	**764**	**9,721**	**393**	**19**

4.2. Labels and features

Weak labels were created for all 9,721 extracted test methods. For each smell, we sampled 100 weak-positive and 100 weak-negative instances, yielding 800 smell-specific validation cases and 778 unique test methods. The sampled instances were curated to estimate label plausibility, and 400 instances were additionally checked by one annotator. We therefore describe the subset as curated validation data rather than fully independent ground truth. Since this check involved a single annotator, inter-rater agreement is not reported. During quality inspection, normalised-token clustering was found to overestimate Duplicate Test for short methods, so a repaired Duplicate Test label was created using a stricter similarity threshold and a minimum token-length constraint. Table 2 reports the validation summary and final training distribution.

Table 2. Curated validation and final label distribution.

Smell	Curated Pos	Agreement Final Pos	Pos	Rate
Assertion roulette	103	83.5%	4,374	45.00%
Eager test	66	83.0%	1,999	20.56%
Mystery guest	73	83.5%	997	10.26%
Duplicate test	80	90.0%	1,858	19.11%

The resulting feature matrix has 9,721 rows and 88 grouped static features. Metadata, labels, and source-code columns are removed before training. Missing feature values are estimated from the training portion of each split. Numerical features are standardised for the linear models, while tree-based models are trained without scaling.

4.3. Baselines, metrics and implementation

We evaluate three groups of outputs: the label-rule consistency check, conventional feature-based models, and stacking. The label-rule consistency check is retained only to verify agreement with the constructed weak labels and is not treated as a fair independent detector baseline. The conventional models are Logistic Regression, Linear SVM, Random Forest, and XGBoost. The stacking ensemble uses the prediction probabilities of the base models as meta-features and Logistic Regression as the meta-learner. All models are trained separately for each smell using the same splits. The implementation uses Python, scikit-learn, and XGBoost with fixed experimental scripts. No held-out test project is used for model selection. F1-score and Macro-F1 are used as the main metrics, and MCC is included to account for class imbalance. Precision and Recall are reported for the cross-project XGBoost analysis. The experimental scripts use fixed data splits and fixed random seeds for stochastic learners; the evaluated versions of JavaParser, scikit-learn, and XGBoost are recorded with the replication material. Because the study contains only three held-out projects, these settings are reported for reproducibility rather than to support any statistical-superiority claim.

5. Results and discussion

5.1. Overall performance

Table 3 reports the average performance of the evaluated methods. The rule-based baseline obtains very high scores because the labels are constructed from weak rules and subsequent repair. Thus, the rule-based row is included only as a label-consistency check. The more informative comparison is among the learned models.

In the within-project setting, Stacking obtains the highest observed learned-model scores, with F1=0.851 and Macro-F1=0.911. XGBoost is close, reaching F1=0.848 and Macro-F1=0.907. This indicates that combining several decision patterns can be useful when training and test instances come from the same project context. In leave-one-project-out evaluation, XGBoost obtains the highest observed learned-model performance, with F1=0.780, Macro-F1=0.861, and MCC=0.733. The larger decrease of Stacking in the cross-project setting suggests that the meta-learner is less stable when trained on only two projects. Because only three held-out project folds are available, these differences should be interpreted descriptively.

Table 3. Overall performance of rule-based and learned models.

Model	W-F1	W-MF1	W-MCC	C-F1	C-MF1	C-MCC
Rule-based	0.974	0.986	0.972	0.977	0.987	0.974
Logistic Reg.	0.780	0.860	0.733	0.668	0.785	0.595
Linear SVM	0.768	0.856	0.722	0.603	0.713	0.505
Random Forest	0.814	0.892	0.787	0.718	0.828	0.689
XGBoost	0.848	0.907	0.815	0.780	0.861	0.733
Stacking	0.851	0.911	0.827	0.694	0.817	0.674

W = Within-project evaluation; C = Cross-project evaluation (leave-one-project-out); F1 = F1-score; MF1 = Macro-F1; MCC = Matthews correlation coefficient.

Overall performance of label-rule consistency and learned models. W denotes within-project evaluation and C denotes leave-one-project-out cross-project evaluation; MF1 denotes Macro-F1. The label-rule consistency row is not an independent detector comparison.

5.2. Per-smell analysis

Table 4 lists the learned model with the highest F1 for each smell. Assertion Roulette and Duplicate Test are more amenable to static feature learning than Mystery Guest. This is expected because assertion structure and token-level similarity are easier to observe at the method level. Duplicate Test reaches very high cross-project performance with XGBoost (F1=0.978), but this result should be read cautiously because the repaired duplicate labels are closely connected to clone-similarity features.

Eager Test is more ambiguous. A long test may still focus on one behaviour, while a shorter test may combine several concerns. Random Forest obtains the highest cross-project F1 for Eager Test (0.695), suggesting that non-linear interactions among call and size features are useful. Mystery Guest is the most difficult smell in the preliminary study, with the highest observed cross-project F1 of 0.568. This supports the need for richer class-level fixture and helper-method analysis.

Table 4. Highest-F1 learned model for each smell.

Smell	Highest within	F1	Highest cross	F1
Assertion Roulette	Stacking	0.935	Stacking	0.895
Eager Test	Stacking	0.859	Random Forest	0.695
Mystery Guest	XGBoost	0.660	XGBoost	0.568
Duplicate Test	Stacking	0.980	XGBoost	0.978

5.3. Leave-one-project-out analysis

The leave-one-project-out setting is the more demanding evaluation because the held-out project is unseen during training. Table 5 shows XGBoost performance averaged across the four smells for each held-out project. Cross-project F1 ranges from 0.744 to 0.826. Commons IO is the easiest held-out case, whereas Commons Collections is more challenging. Because this analysis uses only three held-out projects, the values are reported as descriptive project-level evidence rather than as a basis for confidence intervals or statistical-significance claims. These differences indicate that assertion style, helper methods, and project-specific testing conventions still affect transfer.

XGBoost leave-one-project-out performance averaged over smells (n=3 held-out projects).

Table 5. XGBoost cross-project performance averaged over smells.

Held-out project	Precision	Recall	F1	MCC
Commons collections	0.688	0.830	0.744	0.694
Commons IO	0.874	0.801	0.826	0.766
Commons lang	0.868	0.711	0.771	0.741

Overall, the results support the feasibility of a smell-specific static feature pipeline, while also showing that this preliminary study should not be interpreted as a broad benchmark.

5.4. Feature-group ablation

We performed a compact leave-one-project-out ablation using Random Forest as a diagnostic learner. Table 6 shows the Macro-F1 obtained after removing one feature group at a time. Removing clone-similarity features leads to the largest decrease (-0.040), followed by resource-access features (-0.029). This pattern is consistent with the roles of clone features for Duplicate Test and resource features for Mystery Guest. However, the ablation should be interpreted as diagnostic evidence rather than causal proof or statistical feature-importance analysis. Removing project-normalised features slightly improves Macro-F1, suggesting that such features may encode project-specific style in the small project pool.

Feature-group ablation under leave-one-project-out evaluation. ΔMF1 is computed against the full feature set; results are descriptive for the three-project setting.

Table 6. Feature-group ablation under leave-one-project-out evaluation. ΔMF1 is computed against the full feature set.

Variant	Macro-F1	ΔMF1
Full features	0.815	0.000
w/o assertion	0.808	-0.007
w/o size/complexity	0.811	-0.004
w/o call	0.816	+0.001
w/o fixture	0.815	-0.000
w/o resource	0.786	-0.029
w/o clone	0.775	-0.040
w/o log-transformed	0.814	-0.001
w/o project-normalised	0.831	+0.016

5.5. Threats to validity

Construct validity. The main construct threat comes from the use of weak labels rather than a fully independent expert benchmark. The labelling rules operationalise smell definitions through static source-code signals and may introduce both false positives and false negatives. Learned models may therefore partially approximate the labelling procedure instead of independently recognising test smells. This concern is especially relevant for Duplicate Test, where repaired labels are closely tied to clone-similarity signals. The curated validation subset and secondary inspection reduce this risk, but they do not replace a larger multi-annotator benchmark.

Internal validity. The implementation relies on static parsing, method-level feature extraction, and selected similarity thresholds. Parser configuration, feature normalisation, and missing-value handling may affect final model behaviour. The label-rule consistency row is therefore interpreted only as a sanity check on constructed labels, not as a fair detector baseline. In addition, the current paper reports descriptive comparisons only; no statistical significance test is claimed because the leave-one-project-out design has only three held-out projects.

External validity. The dataset contains only three Apache Java libraries. Although these projects are mature and widely used, the results may not generalise to other ecosystems, industrial projects, Android tests, non-JUnit frameworks, or projects with different testing conventions. The findings should therefore be interpreted as preliminary evidence for

Java/JUnit test-smell screening rather than as a broad benchmark. The existence of dedicated tools for other testing frameworks, such as Pytest-Smell for Python unit tests [18], further highlights the need to evaluate detection approaches across different testing ecosystems.

Conclusion validity. The reported metrics indicate that smell-specific static features can support lightweight screening, especially for smells with observable assertion or similarity signals. However, the evidence is limited by the small project pool, curated weak labels, and method-level approximation of fixture and resource context. Future work should expand the dataset, involve multiple annotators, record confidence intervals over larger project samples, and compare feature-based models with code-language-model baselines under a shared validation protocol.

6. Conclusions

This paper introduced a lightweight feature-engineering pipeline for detecting four JUnit test smells at the method level. The approach combines static test extraction, weak-label generation, curated validation, smell-specific features, and one-vs-rest learning models. In the preliminary three-project evaluation, Stacking performed best in the within-project setting, while XGBoost produced the strongest observed leave-one-project-out results among learned models. The study suggests that feature-based models can support test smell screening, especially for smells with observable assertion or similarity signals. At the same time, the results remain limited by the small project pool, curated weak labels, and method-level approximations of fixture and resource context. Future work will extend the dataset, involve multiple annotators, strengthen class-level extraction, and compare lightweight feature-based methods with code-language-model approaches under a shared validation protocol.

COMPETING INTERESTS

The author declares that there is no conflict of interest regarding the publication of this article.

REFERENCES

[1] D. Campos, L.A. Martins, C.I.M. Bezerra, et al. (2023), “Investigating developers’ contributions to test smell survivability: A study of open-source projects”, *Proceedings of The 8th Brazilian Symposium on Systematic and Automated Software Testing*, pp.41-50, DOI: 10.1145/3624032.3624044.

[2] W.M. Aljedaani, A.S. Peruma, A. Aljohani, et al. (2021), “Test smell detection tools: A systematic mapping study”, *Proceedings of the 25th International Conference on*

Evaluation and Assessment in Software Engineering, pp.170-180, DOI: 10.1145/3463274.3463335.

[3] A. Panichella, S. Panichella, G. Fraser, et al. (2022), "Test smells 20 years later: Detectability, validity, and reliability", *Empirical Software Engineering*, **27**, DOI: 10.1007/s10664-022-10207-5.

[4] D. Campos, L.R. Soares, I. do Carmo Machado (2021), "Developers' perception on the severity of test smells: An empirical study", *arXiv*, https://arxiv.org/abs/2107.13902, accessed 10 June 2026.

[5] M. Marabesi, A. García-Holgado, F.J. García-Peñalvo (2024), "Exploring the connection between the TDD practice and test smells - a systematic literature review", *Comput.*, **13(3)**, DOI: 10.3390/computers13030079.

[6] B. Zhang, P. Liang, X. Zhou, et al. (2024), "A comprehensive evaluation of parameter-efficient fine-tuning on code smell detection", *ACM Transactions on Software Engineering and Methodology*, DOI: 10.1145/3806390.

[7] M.A. Hasan, T. Ahammed (2024), "Understanding the occurrences of test smells in open source and industrial software systems: An empirical study on Python projects", *QuASoQ2024*, https://ceur-ws.org/Vol-3864/quasoq-2024-paper-03.pdf.

[8] E. Soares, M. Aranda, N.D.C. Oliveira, et al. (2023), "Manual tests do smell! Cataloging and identifying natural language test smells", *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp.1-11, DOI: 10.48550/arXiv.2308.01386.

[9] H. Damasceno, C.I.M. Bezerra, E.F. Coutinho, et al. (2022), "Analyzing test smells refactoring from a developer's perspective", *Proceedings of the XXI Brazilian Symposium on Software Quality*, pp.1-10, DOI: 10.1145/3571473.3571487.

[10] R. Rwemalika, S. Habchi, M. Papadakis, et al. (2023), "Smells in system user interactive tests", *Empirical Software Engineering*, **28(2)**, DOI: 10.1007/s10664-022-10251-1.

[11] V. Pontillo, D. Amoroso d'Aragona, F. Pecorelli, et al. (2024), "Machine learning-based test smell detection", *Empirical Software Engineering*, **29**, DOI: 10.1007/s10664-023-10436-2.

[12] R. Melo, P. Simoes, R. Gheyi, et al. (2025), "Agentic LMs: Hunting down test smells", *IEEE Software*, **43**, pp.32-40, DOI: 10.1109/MS.2025.3621356.

[13] S. Souza, A. Santana, E. Figueiredo, et al. (2026), "Beyond strict rules: Assessing the effectiveness of large language models for code smell detection", *arXiv*, DOI:10.48550/arXiv.2601.09873.

[14] M. Mosquera, R. Bojorque, P. Flores (2025), "Understanding code smell detection through hyperparameter optimization and metric correlation analysis", *IEEE Access*, **13**, pp.217750-217768, DOI: 10.1109/ACCESS.2025.2505678.

[15] I. Nakamura, Y. Kashiwa, B. Lin, et al. (2026), "Understanding self-admitted technical debt in test code: An empirical study", *ACM Transactions on Software Engineering and Methodology*, **35(9)**, pp.1-35, DOI: 10.1145/3786791.

[16] H.K.V. Tran, N. bin Ali, M. Unterkalmsteiner, et al. (2025), "Quality attributes of test cases and test suites - importance & challenges from practitioners' perspectives", *Software Quality Journal*, **33**, DOI: 10.1007/s11219-024-09698-w.

[17] M.A. Hasan, T. Ahammed (2025), "Prioritizing test smells: An empirical evaluation of quality metrics and developer perceptions", *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp.815-820, DOI: 10.1109/ICSME64153.2025.00085.

[18] A. Bodea (2022), "Pytest-smell: A smell detection tool for Python unit tests", *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp.793-796, DOI: 10.1145/3533767.3543290.