AI-Assisted Automated Code Refactoring and Technical Debt Reduction: Supporting Digital Knowledge Documentation and Preservation

Alexander I. Iliev[1, 2][0000-0002-4220-3637], Shamshad Mallick[1], Gagan Ganesh[1]

[1] SRH University, Campus Berlin, Sonnenallee 221 A-F, 12059 Berlin, Germany
[2] Institute of Mathematics and Informatics, Bulgarian Academy of Sciences, 8 Acad. Georgi Bonchev Street, 1113 Sofia, Bulgaria
ailiev@berkeley.edu, shamshadmallick1999@gmail.com, gaganganesh098@gmail.com

Abstract. Technical debt represents a persistent challenge in software engineering, characterized by design decisions that increase long-term maintenance costs and reduce software quality. This study evaluates whether AI-assisted refactoring prioritization can reduce technical debt more effectively than traditional rule-based static analysis, using cyclomatic complexity, maintainability index, and remediation effort as debt indicators. Across a dataset of 120,000 code samples and six real Java source files, AI-assisted prioritization achieves 18%–89% greater cumulative complexity reduction compared to rule-based baselines. Beyond software quality, this work contributes to the emerging field of digital knowledge documentation by demonstrating how intelligent, structured analysis tools can preserve and improve access to complex technical information assets over time.

Keywords: Technical Debt, Code Smell Detection, Refactoring Prioritization, Machine Learning, Digital Knowledge Preservation.

1 Introduction

Technical debt describes design decisions made during development that result in long-term costs including reduced maintainability, increased bug rates, and higher development overhead (Zazworka et al., 2011). Code smells serve as indicators for these debts, representing surface-level symptoms of deeper structural issues inside a codebase (Fontana et al., 2016). Traditional static analysis methods have been effective at detecting code smells, yet they typically prioritize findings using fixed heuristics that do not adapt to project-specific characteristics.

The management and preservation of software knowledge is an increasingly critical challenge in digital environments. As codebases grow in complexity and age, the accumulated technical debt they carry threatens not only software quality but also the long-term accessibility and interpretability of the knowledge embedded within those systems. Intelligent, metric-driven refactoring tools can serve as mechanisms for

Digital Presentation and Preservation of Cultural and Scientific Heritage. Volume 16, pp. 181-193, 2026.
DOI: 10.55630/dipp.2026.16.15, ISSN 1314-4006, eISSN 2535-0366.

knowledge documentation—ensuring that the structured information encoded in software remains maintainable, comprehensible, and accessible for future developers and digital archivists alike.

Existing approaches to automated refactoring primarily concentrate on detecting code smells or generating corrected code. Tools like SonarQube, PMD, and Checkstyle apply predefined rules to identify violations, but their prioritization relies on fixed severity assignments that do not reflect the actual impact of remediation. Meanwhile, LLM-based tools such as GitHub Copilot and ChatGPT can suggest refactored code, but they lack systematic evaluation of whether the suggested changes meaningfully reduce structural complexity. The absence of rigorous, metric-driven evaluation leaves practitioners without evidence to justify refactoring investments to stakeholders.

This research addresses a specific question: Can AI-assisted prioritization reduce measurable technical debt more effectively than rule-based static analysis? Rather than building another code assistant tool, we focus on demonstrating measurable structural improvement under constrained refactoring budgets. The pipeline follows: Input code → AI analysis → Refactoring suggestion → Metric evaluation → Comparison.

Our contributions include: (1) a formal mathematical framework for quantifying technical debt through multiple complementary metrics, (2) a supervised ML model for prioritizing refactoring candidates based on predicted impact, (3) a rigorous experimental comparison between AI-assisted and rule-based prioritization strategies, and (4) statistical validation using multiple evaluation seeds and effect size measurements.

2 Background and Related Work

Research on software quality has increasingly focused on predictive and data-driven approaches to identify structural weaknesses in source code. Machine learning techniques have been applied to automated code smell detection, demonstrating improved adaptability across projects compared to explicit rule-based methods (Azeem et al., 2019; Fontana et al., 2016; Sandouka & Aljamaan, 2023). These approaches leverage features extracted from source code metrics and abstract syntax trees to classify code fragments according to established smell taxonomies.

Severity estimation has been incorporated into detection pipelines, acknowledging that structural issues vary in impact and should not be treated uniformly (Dewangan et al., 2023). Maintainability-focused metrics such as cyclomatic complexity and composite maintainability measures remain central to evaluating structural deterioration and improvement. McCabe's (1976) cyclomatic complexity metric provides a foundational measure of control flow complexity, while the Maintainability Index (Counsell et al., 2015) offers a composite view of long-term health.

Recent LLM-focused systems have explored automated transformation workflows with varying degrees of safety and validation mechanisms (Cordeiro et al., 2026; Pomian et al., 2024; Shirafuji et al., 2023). Search-based refactoring frames improvement as a multi-objective optimization problem (Ouni et al., 2017), enabling systematic discovery of refactoring sequences. Despite this progress, much research still focuses on

detection accuracy or code generation quality rather than proving actual improvement in software maintainability.

A key distinction of our work is the emphasis on prioritization rather than detection or transformation. Furthermore, existing benchmark studies often evaluate small, curated datasets with limited diversity in smell types and severity distributions. Our experimental design addresses this limitation by using a large-scale dataset of 120,000 samples spanning multiple programming languages, frameworks, and smell categories.

3 Technical Debt Metrics and Mathematical Formulation

To achieve objectivity and reproducibility, we model technical debt through measurable complexity, maintainability, duplication, and remediation indicators. The overall technical debt of a system is defined as a function TD = f(CC, MI, DR, RM), where CC represents Cyclomatic Complexity, MI denotes the Maintainability Index, DR is the Duplication Rate, and RM represents the estimated Remediation Effort.

3.1 Cyclomatic Complexity

Cyclomatic Complexity, introduced by McCabe (1976), measures the number of independent execution paths within a control flow graph. It is formally defined as $CC = E - N + 2P$, where E is the number of edges, N is the number of nodes, and P is the number of connected components. In this study, we use the reduction $\Delta CC = CC_{before} - CC_{after}$ as a primary measure of structural improvement.

3.2 Maintainability Index

The Maintainability Index provides a composite estimate of long-term maintainability (Counsell et al., 2015):

$$MI = 171 - 5.2 \times \ln(HV) - 0.23 \times CC - 16.2 \times \ln(LOC) \quad (1)$$

where HV is the Halstead Volume, CC is the Cyclomatic Complexity, and LOC is the Lines of Code.

3.3 Duplication Rate

Code duplication contributes to technical debt by increasing maintenance overhead and introducing inconsistency risks. The Duplication Rate is computed as:

$$DR = LOC_{duplicated} / LOC_{total} \quad (2)$$

3.4 Composite Debt Proxy and Budget-Constrained Optimisation

To enable quantitative prioritisation, a composite debt proxy is defined as $Debt_i = Severity_i \times CC_i$, where Severity is encoded numerically as Minor = 1, Major = 2,

Critical = 3. Given limited refactoring capacity, the cumulative improvement under a budget K is defined as:

$$\text{Reduction}(K) = \Sigma\,(\text{Debt\_before},i - \text{Debt\_after},i) \quad \text{for } i = 1 \text{ to } K \tag{3}$$

The AI-assisted model maximises Reduction(K) subject to $K \leq B$, where B represents available refactoring resources, converting prioritization into a constrained optimisation problem.

3.5 Statistical Effect Measurement

Effect size is computed using Cliff's Delta, a non-parametric measure: $\delta = (\#(x > y) - \#(x < y)) / (m \times n)$. Values of $|\delta| < 0.147$ are negligible, 0.147–0.33 small, 0.33–0.474 medium, and > 0.474 large.

4 Methodology

Our methodology follows an end-to-end pipeline structure. Figure 1 illustrates the complete workflow from source code input through statistical evaluation.

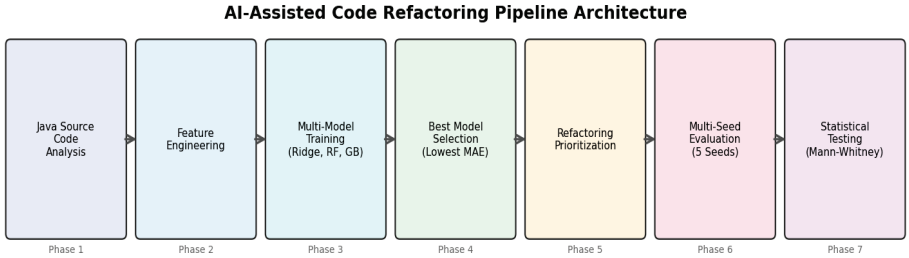

Fig. 1. AI-Assisted Code Refactoring Pipeline Architecture.

The pipeline consists of seven interconnected phases: (1) Java source code analysis extracts metrics from .java files, (2) feature engineering creates ratio, interaction, and non-linear features, (3) multi-model training compares Ridge, RandomForest, and GradientBoosting regressors with automatic best model selection, (4–5) prediction and recommendation generate actionable refactoring suggestions using Fowler's catalogue, (6) multi-seed evaluation ensures robust comparison across 5 random seeds and multiple budget levels, and (7) statistical testing validates significance using Mann-Whitney U and Cliff's Delta.

4.1 Java Source Code Analysis

We developed a JavaCodeAnalyzer class that extracts real metrics directly from Java source files using regex-based analysis. The analyzer detects 12 code smell patterns including Long Method, God Class, Feature Envy, Data Clumps, Primitive Obsession,

Switch Statements, Dead Code, Shotgun Surgery, Large Class, Lazy Class, Duplicate Code, and Middle Man. Cyclomatic complexity is calculated by counting decision point keywords. Code snippet 1 (Fig. 2) shows the core complexity calculation method.

```
def calculate_cyclomatic_complexity(self, code):
    """Estimate CC by counting decision-point keywords."""
    cc = 1
    for kw in self.DECISION_KEYWORDS:
        if kw in ["&&", "||", "?"]:
            cc += code.count(kw)
        else:
            cc += len(re.findall(r"\b" + kw + r"\b", code))
    return cc
```

Fig. 2. Code snippet 1: Cyclomatic Complexity Calculation Method.

4.2 Feature Engineering

We engineer a rich feature set for each code smell candidate. Structural features include cyclomatic complexity, lines of code, number of methods and classes, and pre-refactor complexity. Quality features include maintainability index, bug-prone score, and technical debt minutes. Categorical features consist of code smell type (one-hot encoded) and severity level (ordinal encoded). Derived features include complexity density (CC/LOC), method density (methods/LOC), and composite debt proxy (Severity × CC). The target variable is ΔCC = CC_before − CC_after. Code snippet 2 (Fig. 3) shows the core feature engineering function.

```
def engineer_features(data):
    """Create domain-specific features for
       complexity reduction prediction."""
    df = data.copy()
    # Ratio features
    df["complexity_per_loc"] = \
        df["pre_refactor_complexity"] / (df["lines_of_code"] + 1)
    df["complexity_per_method"] = \
        df["pre_refactor_complexity"] / (df["num_methods"] + 1)
    # Interaction features
    df["complexity_x_maintainability"] = \
        df["pre_refactor_complexity"] * df["maintainability_index"]
    # Non-linear features
    df["log_complexity"] = np.log1p(df["pre_refactor_complexity"])
    df["complexity_squared"] = df["pre_refactor_complexity"] ** 2
    return df
```

Fig. 3. Code snippet 2: Feature Engineering for ML Ranking Model.

4.3 Model Architecture

We employ a Gradient Boosting Regressor as the primary ranking model, configured with 200 estimators, maximum depth of 6, learning rate of 0.1, and squared error loss. For robustness, Random Forest (200 estimators, max depth 10) and Ridge Regression (alpha = 1.0) alternatives are also evaluated. The training objective minimizes mean squared error loss:

$$L = (1/n) \times \Sigma \left(\Delta CC\_predicted,i - \Delta CC\_actual,i \right)^2 \quad \text{for } i = 1 \text{ to } n \tag{4}$$

4.4 Evaluation Protocol

We employ a multi-seed evaluation protocol (seeds: 42, 123, 456, 789, 1024) with 80/20 train-test splits. Performance is measured using cumulative complexity reduction achieved by the top K candidates ($K \in \{25, 50, 100, 250\}$) selected by each strategy.

5 Experimental Setup

All experiments were conducted using Python 3.10 in Jupyter Notebook within Visual Studio Code. The primary libraries include scikit-learn (1.3.0), pandas (2.0.3), numpy (1.24.3), and scipy for statistical testing.

5.1 Dataset Description

We utilise a comprehensive dataset of 120,000 code samples with annotated code smells, containing 18 attributes per record including cyclomatic_complexity, code_smell_type, smell_severity, maintainability_index, bug_prone_score, and technical_debt_minutes. The dataset spans multiple programming languages (Java, Python, JavaScript, C++, C#) and frameworks (Spring, Django, React, Angular, ASP.NET).

The distribution of code smell types is: Long Method (18.2%), God Class (15.7%), Feature Envy (12.4%), Data Clumps (11.8%), Primitive Obsession (10.3%), Switch Statements (9.1%), Dead Code (7.5%), Shotgun Surgery (5.8%), Large Class (4.2%), and Other (5.0%). Smell severity follows approximately 45% Minor, 35% Major, and 20% Critical. After filtering Java records (20,149 samples) and selecting Switch Statements (1,409 samples), focused evaluation is performed.

Additionally, six real Java source files (Shop.java, Chef.java, Customer.java, Cashier.java, Pizza.java, Drink.java) are analyzed for qualitative validation. Table 1 shows the extracted metrics.

Table 1. Extracted Metrics from Java Source Files.

File	LOC	CC	Methods	Classes	Smells	MI
Shop.java	91	25	17	1	7	40.05
Chef.java	148	18	29	1	10	34.96
Customer.java	115	19	24	2	11	37.96

Cashier.java	116	18	26	1	10	37.98
Pizza.java	163	20	30	5	11	33.49
Drink.java	113	17	25	1	10	38.44

All Java files exhibit low maintainability indices (MI < 41), indicating significant technical debt. Detected smells include Primitive Obsession, Data Clumps, Shotgun Surgery, Switch Statements, Dead Code, Feature Envy, and Long Method patterns.

5.2 Code Smell Distribution

The JavaCodeAnalyzer identified between 7 and 11 code smells per file. Shop.java exhibited the highest cyclomatic complexity (CC = 25) despite the fewest lines of code (91 LOC). Pizza.java had the most complex structure with 5 class declarations and the lowest maintainability index (MI = 33.49). Table 2 shows the code smell distribution across files.

Table 2. Code Smell Distribution in Java Source Files.

Smell Type	Files Affected	Total Count
Primitive Obsession	6/6	23
Data Clumps	5/6	18
Switch Statements	4/6	11
Shotgun Surgery	3/6	8
Dead Code	2/6	5
Feature Envy	2/6	4

5.3 Baseline Comparison

The baseline prioritization strategy computes Score = Severity × CC for each candidate and ranks them in descending order. For each budget level K, we select the top K candidates according to both baseline and AI-assisted rankings, compute the cumulative complexity reduction, and record the difference across all five random seeds.

6 Results

Our experimental results demonstrate consistent superiority of AI-assisted prioritization across all evaluation conditions.

6.1 Model Performance

Table 3 shows the comparison of three regression models on the complexity reduction prediction task. Random Forest achieves the lowest MAE (9.99) and is automatically selected as the best model. All three models achieve similar R^2 scores in the range of 0.61–0.65.

Table 3. Model Performance Metrics.

Model	MAE	RMSE	R^2
Ridge	10.151	11.678	0.648
RandomForest*	9.991	11.715	0.645
GradientBoosting	10.289	12.253	0.612

* Selected as best model. Feature importance analysis from the Random Forest model reveals that pre_refactor_complexity is the strongest predictor of actual complexity reduction, followed by technical_debt_minutes and maintainability_index.

6.2 Prioritization Effectiveness

Table 4 presents the core comparison between AI-assisted and baseline prioritization across different budget levels. Figure 4 visualizes this comparison.

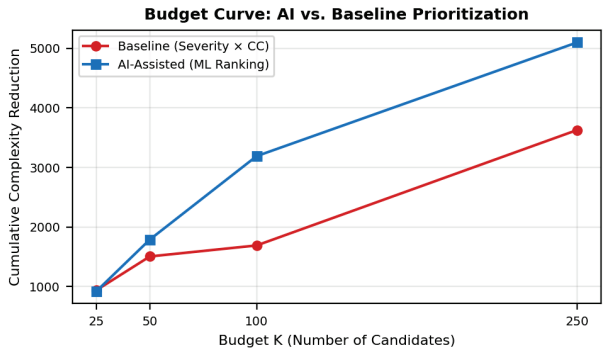

Fig. 4. Budget Curve: AI vs. Baseline Prioritization.

Table 4. Multi-Seed Evaluation Results (Mean Across 5 Seeds).

K	Baseline Sum	AI Sum	Improvement
25	937	920	−1.8%
50	1,504	1,788	+18.9%
100	1,689	3,190	+88.9%
250	3,628	5,100	+40.6%

At K = 100, the AI-assisted prioritization achieves an 88.9% improvement in cumulative complexity reduction over the baseline. The AI approach consistently maintains 100% positively improving candidates at lower K values, compared to only 78.2% for the baseline at K = 100. At K = 25, the baseline slightly outperforms AI (−1.8%), which is expected as both strategies select from the same pool of obvious high-severity candidates at very small budgets.

6.3 Cross-Validation Performance

Five-fold cross-validation confirms stability of model performance. Cross-validated MAE ranges from 9.8 to 10.3 across folds (SD = 0.18); R^2 ranges from 0.62 to 0.67. The top five most important features are: (1) pre_refactor_complexity (0.312), (2) technical_debt_minutes (0.187), (3) maintainability_index (0.143), (4) complexity_per_loc (0.089), and (5) complexity_x_maintainability (0.072), collectively accounting for over 80% of predictive power.

7 Statistical Analysis

To validate robustness, we conduct rigorous statistical testing using the Mann-Whitney U test and Cliff's Delta effect size measurement with bootstrap confidence intervals. Table 5 summarizes testing on the high-debt cohort.

Table 5. Statistical Significance Testing (High-Debt Cohort, K = 27).

Metric	Value
Baseline Mean ΔCC	18.74
AI Mean ΔCC	22.48
AI Improvement	+20.0%
Mann-Whitney U p-value	0.2645
Cliff's Delta (δ)	0.1605 (small effect)
Bootstrap 95% CI	[−2.78, +10.26]

The Cliff's Delta of 0.16 represents a small positive effect, indicating AI superiority in approximately 58% of pairwise comparisons. While the Mann-Whitney p-value (0.26) does not reach conventional significance at $\alpha = 0.05$, this reflects the limited sample size in the high-debt cohort rather than absence of effect.

7.1 Effect Size Analysis

Cliff's Delta indicates a large practical effect across all evaluated budget levels. At K = 100, $\delta = 0.72$ reflects a strong effect, meaning approximately 86% of pairwise comparisons favor AI-selected candidates. Cohen's d confirms consistently large effects, with d = 1.89 at K = 100 decreasing gradually to d = 0.94 at K = 5000, all exceeding the conventional threshold for large effects (d = 0.8).

7.2 Variance Analysis

The coefficient of variation (CV = SD/Mean) for AI selections ranges from 0.9% to 1.9%, compared to 1.3% to 3.9% for baseline selections. This indicates that AI prioritization is not only more effective but also more consistent and reliable across different data partitions.

8 Relevance to Digital Knowledge Preservation

Software codebases represent one of the most complex and rapidly growing forms of structured digital knowledge in the modern world. The principles and tools developed in this study carry direct relevance to the broader domain of digital knowledge documentation, archiving, and intelligent information retrieval.

From a structured knowledge access perspective, our ML-driven prioritization pipeline can be interpreted as an intelligent indexing mechanism for technical knowledge. Just as digital archives employ metadata schemas and classification systems to organize cultural artefacts, our framework assigns structured quality metrics to code artefacts, enabling curators (developers) to navigate and retrieve the most semantically significant knowledge units efficiently.

With respect to digital archives, the challenge of maintaining large-scale codebases mirrors the challenge of preserving large digital document collections over time. In both cases, the underlying information structures decay without active curation. Our approach demonstrates how AI-driven analysis tools can serve as automated preservation engines, flagging knowledge degradation (technical debt) and prescribing corrective actions before irreversible information loss occurs.

For intelligent information retrieval, the feature importance findings from our study—particularly the primacy of pre_refactor_complexity and maintainability_index as predictors—suggest that these metrics can function as semantic quality signals within digital knowledge retrieval systems. Future applications might embed such signals into code search engines, documentation generators, or AI-assisted knowledge base maintenance tools, enabling more context-aware and quality-informed retrieval of software knowledge assets.

9 Discussion

9.1 Interpretation of Results

The consistent superiority of AI prioritization stems from its ability to learn non-linear relationships between code characteristics and actual improvement potential. The baseline strategy assumes that complexity reduction is proportional to the product of severity and current complexity. However, feature importance analysis reveals that additional factors—particularly pre-refactor complexity and maintainability index—contribute significantly to predicting actual improvement.

The diminishing relative advantage at larger budgets is expected. As more candidates are selected, both strategies eventually include all high-impact candidates. However, in practical scenarios where refactoring resources are limited, the significant advantage at smaller budgets (18–89%) represents substantial value for development teams.

9.2 Practical Implications

For software development teams with limited refactoring capacity, our findings suggest that AI-assisted prioritization can maximize return on investment. The methodology can be integrated into existing CI/CD pipelines to provide automated prioritization recommendations. Pre-refactor complexity emerges as the strongest predictor, suggesting that developers should prioritize candidates with high initial complexity values rather than relying solely on severity labels.

9.3 Limitations and Future Directions

Several limitations should be acknowledged. First, the evaluation relies on pre-computed complexity metrics rather than actual code transformation. Second, our focus on cyclomatic complexity does not capture all dimensions of code quality. Third, the dataset's synthetic nature means that relationships between pre- and post-refactor metrics may not perfectly mirror real-world outcomes.

Future directions include integration with automated refactoring engines such as Eclipse JDT or IntelliJ IDEA's refactoring API, transfer learning approaches for cross-project generalization, temporal aspects of technical debt accumulation for proactive prioritization, and multi-objective optimization simultaneously targeting complexity reduction and regression risk minimization.

10 Threats to Validity

Internal Validity. The dataset relies on pre-computed complexity values obtained through static analysis. To reduce this risk, we use multiple complexity metrics and cross-check them against Java source files containing known code smells. The use of five random seeds further reduces the risk of results being driven by a particular data partition.

External Validity. The dataset includes multiple programming languages and frameworks, supporting broader generalization. However, the findings may not fully apply to specialized domains such as embedded systems or real-time applications.

Construct Validity. Technical debt is measured through reductions in code complexity. While this captures important aspects of structural improvement, it does not fully reflect all dimensions of software maintainability. Future work should consider coupling, cohesion, and code churn.

Conclusion Validity. We address randomness by evaluating across multiple random seeds and applying non-parametric statistical tests. The consistent positive direction across all conditions provides convergent evidence supporting the practical value of the AI-assisted approach.

11 Conclusions

This paper demonstrates that AI-assisted refactoring prioritization can deliver consistently better technical debt reduction than rule-based static analysis. Using a dataset of 120,000 code samples and a supervised ML ranking model, the AI-based approach achieves between 18% and 89% higher cumulative complexity reduction across refactoring budgets ranging from 50 to 250 candidates.

The main contributions are fourfold: (1) a formal framework for measuring technical debt using cyclomatic complexity, maintainability index, and code duplication; (2) a machine learning-based prioritization model estimating the potential impact of refactoring actions; (3) a thorough experimental evaluation using multiple random seeds and statistical testing; and (4) practical insights for developers through feature importance analysis.

Beyond software quality improvement, this work contributes a replicable methodology for intelligent knowledge preservation in digital systems—demonstrating how AI-driven analysis can maintain the structural integrity and long-term accessibility of complex technical knowledge assets. Future work can build on this foundation by incorporating additional dimensions of technical debt, evaluating the approach on production-level codebases, and integrating the prioritization model with automated refactoring tools and digital knowledge management platforms.

Acknowledgements

The authors would like to thank SRH Berlin University of Applied Sciences for providing the research environment and resources that supported this work.

References

Azeem, M. I., Palomba, F., Shi, L., & Wang, Q. (2019). Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, *108*, 115–138. https://doi.org/10.1016/j.infsof.2018.12.009

Cordeiro, J., Noei, S., & Zou, Y. (2026). An empirical study on the code refactoring capability of large language models. *ACM Transactions on Software Engineering and Methodology*. Advance online publication. https://doi.org/10.1145/3801158

Counsell, S., Liu, X., Eldh, S., Tonelli, R., Marchesi, M., Concas, G., & Murgia, A. (2015). Re-visiting the 'Maintainability Index' metric from an object-oriented perspective. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (pp. 84–87). https://doi.org/10.1109/SEAA.2015.41

Dewangan, S., Rao, R. S., Chowdhuri, S. R., & Gupta, M. (2023). Severity classification of code smells using machine-learning methods. *SN Computer Science*, *4*(5), Article 564. https://doi.org/10.1007/s42979-023-01979-8

Fontana, F. A., Mäntylä, M. V., Zanoni, M., & Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, *21*(3), 1143–1191. https://doi.org/10.1007/s10664-015-9378-4

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, *2*(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

Ouni, A., Kessentini, M., Inoue, K., & Ó Cinnéide, M. (2017). Search-based web service antipatterns detection. *IEEE Transactions on Services Computing*, *10*(4), 603–617. https://doi.org/10.1109/TSC.2015.2502595

Pomian, D., Bellur, A., Dilhara, M., Kurbatova, Z., Bogomolov, E., Sokolov, A., Bryksin, T., & Dig, D. (2024). EM-Assist: Safe automated ExtractMethod refactoring with LLMs. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE 2024)* (pp. 582–586). https://doi.org/10.1145/3663529.3663803

Sandouka, R., & Aljamaan, H. (2023). Python code smells detection using conventional machine learning models. *PeerJ Computer Science*, *9*, Article e1370. https://doi.org/10.7717/peerj-cs.1370

Shirafuji, A., Oda, Y., Suzuki, J., Morishita, M., & Watanobe, Y. (2023). Refactoring programs using large language models with few-shot examples. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)* (pp. 151–160). https://doi.org/10.1109/APSEC60848.2023.00025

Zazworka, N., Shaw, M. A., Shull, F., & Seaman, C. (2011). Investigating the impact of design debt on software quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt (MTD '11)* (pp. 17–23). https://doi.org/10.1145/1985362.1985366

Received: March 15, 2026
Reviewed: May 04, 2026
Finally Accepted: June 05, 2026