

FROM CODE SMELL DETECTION TO REFACTORING RECOMMENDATION: AN EXPLAINABLE RELATION-AWARE TRANSFORMER FRAMEWORK

Israr Ali^{*1}, Ali Ahmed Siddiqui², Aarij Mahmood Hussaan³

^{*1,2,3}Department of Computer Science, Iqra University, Karachi, Pakistan

^{*1}Israr.ali@iqra.edu.pk, ²alisiddiqui@iqra.edu.pk, ³aarijhussaan@gmail.com

DOI: <https://doi.org/10.5281/zenodo.22806551>

Keywords

Software refactoring; Code smells;
Refactoring recommendation;
Relation-aware BERT;
Explainable AI; Software metrics;
Software maintainability

Article History

Received: 01 June 2026

Accepted: 23 August 2026

Published: 17 September 2026

Copyright @Author

Corresponding Author: *

Israr Ali

Abstract

Software refactoring improves maintainability without changing external behavior, but many intelligent code smell studies stop at detecting problematic code and provide limited support for selecting actionable refactoring operations. Objectives: This study investigates RABERT-RR, an explainable relation-aware transformer for predicting candidate refactoring types from code and software metrics. Recommendations are decision-support outputs and are not validated executable transformations. The framework combines CodeBERT-based source-code embeddings with normalized software metrics and a relation-aware metric layer that models dependencies among size, complexity, coupling, cohesion, Halstead, and maintainability indicators. A fusion classifier predicts refactoring operations, while SHAP and LIME explain feature influence. The model was evaluated on a Java multi-class refactoring recommendation dataset of 11,072 samples using a 70:15:15 train-validation-test split and compared with classical machine learning, deep learning, transformer-based, and LLM-based baselines. The reported experiment on 11,072 Java instances gives 0.913 accuracy and 0.902 F1-score for RABERT-RR. These headline results require reconciliation with the per-class aggregates before definitive comparative conclusions can be drawn. The existing quality summaries mix historical observations and structural estimates; they do not establish behavior-preserving improvements caused by the recommendations. Combining code representations with inter-metric relationships is a promising approach to refactoring-type recommendation. Establishing practical effectiveness requires executable transformation validation, reconciled prediction-level results, and a publicly archived replication package. This revision specifies the required validation and release protocols; it does not report their completion.

1. INTRODUCTION

Software systems continuously evolve in response to changing requirements, technology upgrades, bug fixes, and new features. During this evolution,

source code often becomes more complex because new functionality is added under delivery pressure and without sufficient restructuring. This gradual design degradation increases maintenance cost, reduces readability, weakens modularity, and

contributes to technical debt. Refactoring addresses this problem by restructuring existing source code to improve internal design while preserving externally observable behavior [1], [2]. Code smells are widely used as indicators of refactoring opportunities. Smells such as Large Class, Long Method, Feature Envy, Data Class, complex conditional logic, duplicated behavior, and inheritance-related problems signal possible violations of modularity, cohesion, coupling, encapsulation, or responsibility assignment principles [2]-[5]. Detecting a smell, however, is only the first step. In practical maintenance, developers also need to decide which refactoring operation is suitable, why it is appropriate, and whether it is likely to improve quality.

A substantial body of work has investigated rule-based, metric-threshold, machine learning, deep learning, and transformer-based code smell detection [6]-[12]. These approaches have improved automated identification of problematic code, but many of them stop at a smell label or a binary classification result. Traditional refactoring recommendation systems often rely on thresholds, expert rules, or search-based optimization [9], [10], which are interpretable but can be sensitive to project context and difficult to generalize. Transformer-based code models such as CodeBERT and GraphCodeBERT capture semantic source-code patterns, yet they do not explicitly model the metric relationships that frequently determine refactoring suitability [11], [12].

Refactoring is not only a code-token understanding task. A recommendation such as Extract Class is usually supported by a combination of high lines of code, high method count, high attribute count, low cohesion, and high coupling. Extract Method is often associated with long method length, high cyclomatic complexity, nested conditionals, and high Halstead difficulty. Move Method depends on interactions between external data access, coupling, cohesion, and class responsibility. Treating these metrics as isolated numeric attributes may therefore miss meaningful design relationships.

Another important concern is explainability. Developers may be reluctant to apply automated refactoring recommendations if a tool cannot explain why a specific operation was suggested. Incorrect refactorings can break dependencies, alter responsibilities, introduce regression faults, or reduce architectural consistency [13], [14]. Explainable AI methods such as SHAP and LIME can help make recommendations more transparent by identifying which metrics, code features, or structural relationships most influenced a prediction [15], [16].

This study proposes RABERT-RR for recommending candidate refactoring types, including Extract Method, Extract Class, Move Method, Rename Method, Encapsulate Field, Extract Interface, Pull Up Method, Push Down Method, Decompose Conditional, and Replace Conditional with Polymorphism. The model predicts an operation category; it does not determine all transformation arguments, execute a patch, or certify semantic equivalence. Historical or estimated quality indicators are exploratory evidence only.

The main contributions of this study are as follows:

- An explainable relation-aware transformer framework, RABERT-RR, is proposed for refactoring recommendation after code smell detection.
- The framework integrates semantic source-code embeddings with software metrics and explicit inter-metric relationship modeling.
- The study evaluates multi-class refactoring recommendation using classical machine learning, deep learning, transformer, and LLM-based baselines.
- SHAP and LIME are incorporated to expose influential input features; developer trust and explanation usefulness remain to be evaluated.
- The study distinguishes recommendation accuracy, exploratory structural-quality indicators, and behavior-preservation evidence, and specifies an executable validation protocol and a replication-package release contract.

The remainder of the paper is organized as follows. Section 2 reviews related work. Section 3 presents

the RABERT-RR framework. Section 4 describes the experimental setup. Section 5 reports and discusses the results. Section 6 presents threats to validity, and Section 7 concludes the paper.

Research Gap from Code Smell Detection to Refactoring Recommendation

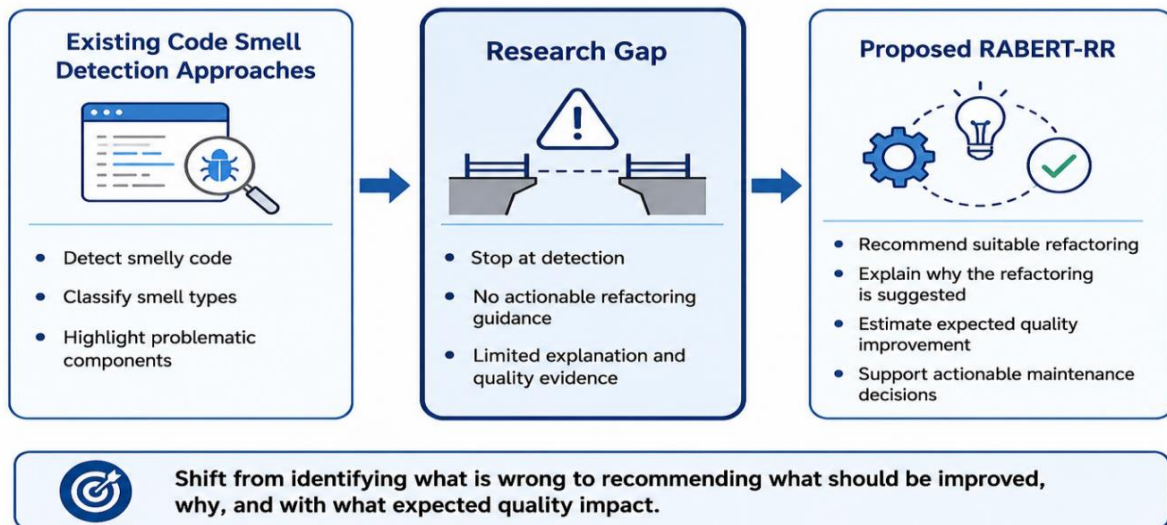


Fig. 1. Research gap from code smell detection to explainable refactoring recommendation. AI-assisted schematic reviewed by the authors.

2. Related work

2.1 Software refactoring, code smells, and repository mining

Refactoring was formalized by Opdyke as a set of behavior-preserving transformations for object-oriented systems [1] and later popularized through Fowler's catalog of practical refactorings [2], [4]. Refactoring is now central to software maintenance because it helps reduce design degradation, improve readability, and manage technical debt without changing external behavior. Mens and Tourwe reviewed the foundations and challenges of software refactoring, emphasizing that tool support must balance automation, correctness, and developer control [3].

Code smells and refactoring are closely linked. Lacerda et al. reported that code smell detection is frequently used to identify refactoring opportunities, but the relationship between a

smell and a suitable refactoring is not one-to-one [5]. For example, Large Class may be addressed through Extract Class, Move Method, or Extract Interface depending on responsibility distribution and dependency structure. Tufano et al. further showed that smells are often introduced early in a code artifact's life rather than only after long-term decay [8], which supports continuous and proactive refactoring support.

Repository mining enables the construction of empirical refactoring datasets. RefactoringMiner detects refactoring operations from commit histories and has been widely used in empirical studies [21], while RefDiff provides multi-language refactoring detection support [22]. Repository mining also provides insight into developer motivation. Silva et al. found that developers refactor not only to remove smells but also to improve readability, accommodate changing requirements, and restructure design [23]. These

findings suggest that recommendation systems should combine smell evidence, code semantics, metrics, and quality impact rather than treating refactoring as a direct smell-to-label mapping.

2.2 Machine learning, deep learning, transformers, and LLMs

Machine learning approaches usually rely on software metrics such as LOC, cyclomatic complexity, coupling, cohesion, Halstead measures, and maintainability indicators. Metric-based models are useful because their inputs are interpretable, but they can miss semantic code patterns and are sensitive to project-specific thresholds. Search-based approaches can optimize refactoring sequences with respect to multiple objectives [10], yet they often require manually designed fitness functions and may provide limited explanation to developers.

Deep learning methods learn representations directly from code sequences. Nyirongo et al. surveyed deep learning-based refactoring research and noted that many studies focus on smell detection or recommendation, while fewer address behavior preservation, quality validation, reproducibility, and explainability [25]. Sequence models such as CNN, LSTM, and BiLSTM capture local and sequential code patterns, but they are weaker at long-range dependencies and structural relationships.

Transformer-based code models address some of these limitations. CodeBERT learns representations from programming and natural languages and supports code intelligence tasks [11]. GraphCodeBERT incorporates data-flow information into code representation learning [12]. PLBART uses a sequence-to-sequence denoising objective for program understanding and generation [26]. These models provide strong code semantics, but refactoring decisions also depend on quality-related structural indicators. A transformer-only representation may learn what code does without fully capturing why a refactoring is suitable.

Large language models have recently been applied to code generation, repair, testing, summarization, and maintenance [27]. LLM-based refactoring is

promising because models can generate natural-language explanations and proposed code transformations. However, systematic reviews indicate unresolved challenges related to correctness, prompt sensitivity, reproducibility, project-level context, metric-awareness, and behavior preservation [28], [29]. In this study, an LLM-based recommender is therefore used as a baseline, while the proposed approach focuses on controlled, metric-aware, and reproducible classification.

2.3 Explainability, quality impact, and research gap

Empirical work shows that refactoring does not automatically improve every software quality attribute. Cedrim et al. investigated structural quality across multiple projects and questioned the assumption that refactoring always reduces smell density [17]. Kaur and Singh also reported that refactoring can affect quality attributes in different ways; for example, Extract Method may reduce method complexity while increasing method calls, and Extract Class may improve cohesion while introducing additional dependencies [18]. A useful recommendation system should therefore be evaluated not only by classification metrics but also by quality-impact indicators.

Explainability is crucial because refactoring decisions affect program structure. SHAP quantifies feature contributions using additive explanations [15], while LIME approximates model behavior around individual instances [16]. In refactoring recommendation, explanations should align with accepted software engineering principles. For example, Extract Class should be justified through responsibility-related signals such as high LOC, high method count, low cohesion, and high coupling. Such explanations can help developers validate recommendations before applying them.

The reviewed literature motivates combining code semantics, structural metrics, and explanation when recommending refactoring types. RABERT-RR investigates this combination. Its scope does not resolve transformation correctness or

reproducibility by architecture alone: executable validation evidence and a released package are necessary to evaluate those properties.

3. RABERT-RR framework

3.1 Overview

RABERT-RR predicts a candidate refactoring type from a code component and associated structural evidence. Its seven phases are dataset preparation, source-code and metric extraction, code representation, relation-aware metric representation, feature fusion, classification, and explanation with exploratory quality analysis. A

separate validation stage is required before a recommendation may be treated as an applicable transformation; Section 3.5 specifies that stage.

Each instance contains a source-code component C , a software metric vector M , a code smell label S , and a target refactoring label R . The model learns from both semantic source-code information and structural quality evidence. The input representation can be expressed as editable text as follows:

$x = \{C, M, S\}$, where C is the code component, M is the metric vector, and S is the detected smell. (1)

Overall Architecture of RABERT-RR

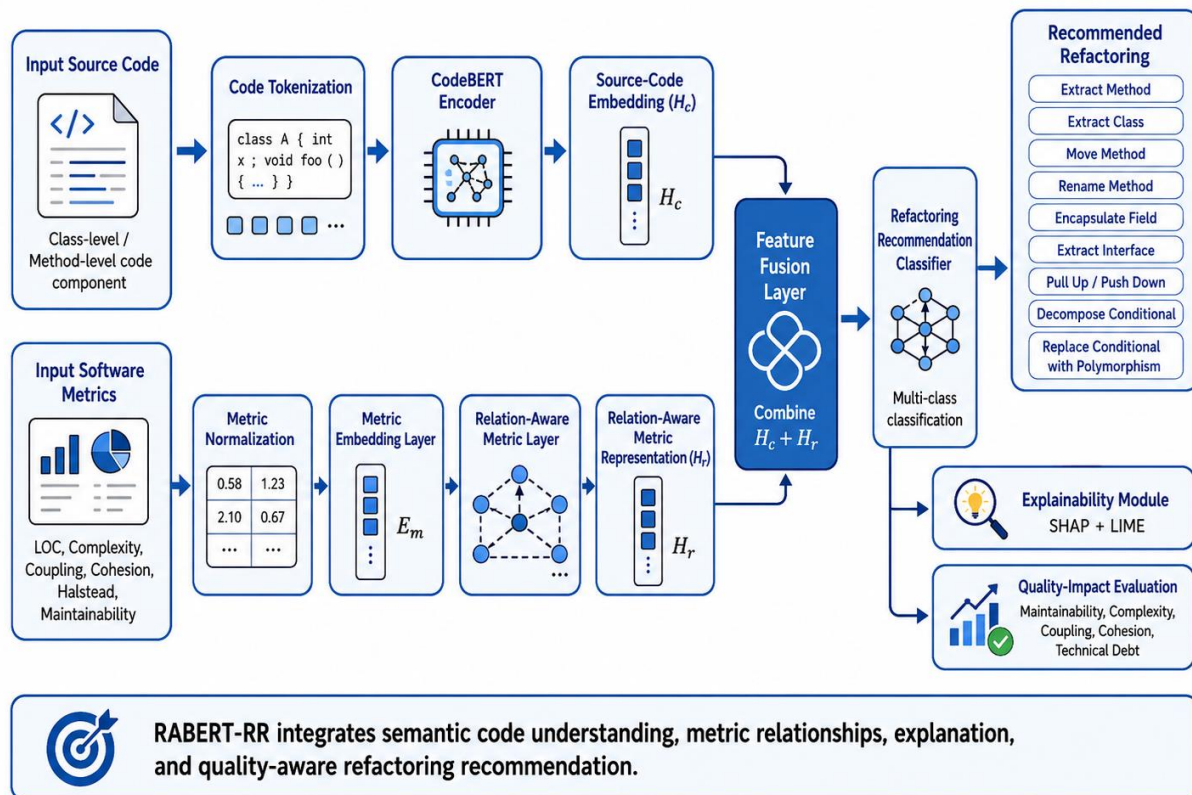


Fig. 2. Overall architecture of RABERT-RR. AI-assisted schematic reviewed by the authors.

3.2 Dataset preparation and label construction

The dataset was prepared from Java-based open-source object-oriented repositories because Java has mature refactoring-mining and metric-extraction support. Refactoring operations were

mined from commit histories using RefactoringMiner [21]. Code smell labels were obtained from available smell labels and metric-based detection rules. Mined refactorings were aligned with changed class-level or method-level

code components to construct supervised labels for recommendation.

The smell-refactoring mapping in Table 1 was used to support label construction. Because one smell can correspond to more than one valid operation, the present version formulates the task as multi-

class classification and treats multi-label and sequence-based recommendation as future work. Duplicate fragments, incomplete entries, and ambiguous low-confidence mined refactorings were removed or manually checked to reduce noise.

Table 1. Smell-refactoring mapping used in the proposed framework.

Code smell	Possible refactoring operations
Large Class	Extract Class; Move Method; Extract Interface
Long Method	Extract Method; Decompose Conditional
Feature Envy	Move Method
Data Class	Encapsulate Field; Move Method
Complex Conditional	Decompose Conditional; Replace Conditional with Polymorphism
Inheritance-related design issue	Pull Up Method; Push Down Method; Extract Interface
Naming inconsistency	Rename Method

Table 2. Dataset summary and experimental split.

Property	Description
Programming language	Java
Source type	32 mature open-source GitHub repositories
Refactoring mining tool	RefactoringMiner
Code smell categories	Large Class; Long Method; Feature Envy; Data Class; Complex Conditional; inheritance-related design issue; naming inconsistency
Refactoring classes	Extract Method; Rename Method; Extract Class; Move Method; Encapsulate Field; Decompose Conditional; Extract Interface; Pull Up Method; Push Down Method; Replace Conditional with Polymorphism
Total samples	11,072 code component instances
Training / validation / test	7,750 / 1,661 / 1,661
Split ratio	70:15:15
Feature types	Source-code embeddings and software metrics
Prediction task	Multi-class refactoring recommendation

Dataset provenance remains incomplete in this revision: the exact repository list, commit identifiers, licenses, sample manifest, and split membership have not been supplied with the manuscript. The totals in Table 2 are retained as reported and must be verified against the released records.

3.3 Feature extraction and relation-aware modeling

Two feature groups are used. Source-code features include class names, method names, attribute names, method bodies, control-flow statements, API calls, object interactions, inheritance information, and semantically useful comments. The code component is tokenized using the

selected transformer tokenizer. Long fragments are truncated or segmented, while shorter fragments are padded.

Software metrics capture structural and quality-related properties that are directly relevant to refactoring decisions. Table 3 summarizes the metric categories. Raw metrics are normalized

using z-score normalization to reduce scale dominance by high-magnitude metrics such as LOC or Halstead effort. The code-token sequence and metric vector are represented as:

$$C = [c_1, c_2, \dots, c_n]. \quad (2)$$

$$M = [m_1, m_2, \dots, m_k]. \quad (3)$$

Table 3. Software metric categories used in RABERT-RR.

Metric category	Metrics
Size	LOC; logical LOC; number of methods; number of attributes; number of statements
Complexity	Cyclomatic complexity; weighted methods per class; nesting depth
Halstead	Vocabulary; length; volume; difficulty; effort; estimated bugs
Object-oriented	Coupling Between Objects; Lack of Cohesion in Methods; Depth of Inheritance Tree; Number of Children; Response for a Class
Maintainability	Maintainability Index; code smell density; technical debt indicator

CodeBERT is used as the main transformer encoder because it is pre-trained on programming and natural language data [11]. The final hidden representation of the classification token is used as a compact code embedding:

$$H_c = \text{CodeBERT}(C). \quad (4)$$

The metric vector is first projected into a dense space and then passed through a relation-aware self-attention layer. This layer is intended to model dependencies such as LOC-complexity, coupling-cohesion, method count-responsibility, and complexity-maintainability relationships:

$$H_m = \text{sigma}(W_m M + b_m). \quad (5)$$

$$H_r = \text{RelationAttention}(H_m, H_m, H_m). \quad (6)$$

The code representation and relation-aware metric representation are then combined by a fusion layer:

$$H_f = \text{Fusion}(H_c, H_r). \quad (7)$$

The fused representation is passed to a softmax classifier that predicts the refactoring label:

$$P(R | C, M, S) = \text{softmax}(W_o H_f + b_o). \quad (8)$$

$$L = - \sum_i y_i \log(\hat{y}_i). \quad (9)$$

3.4 Explainability and quality-impact evaluation

RABERT-RR includes an explainability module to help developers understand why a refactoring was recommended. SHAP is used to estimate global and local feature contributions, and LIME is used to generate local explanations for selected instances [15], [16]. For example, an Extract Class recommendation should highlight excessive class responsibility indicators such as high LOC, high method count, high attribute count, low cohesion, and high coupling. An Extract Method recommendation should highlight method length, cyclomatic complexity, nesting depth, and Halstead difficulty.

The original quality analysis combines metrics from historical post-change versions with structurally estimated values when executable post-change versions are unavailable. These evidence types must be separated. A historical commit may include unrelated edits, and an estimated metric change is not a measured transformation outcome. Table 11 is therefore retained only as an unverified exploratory summary. For the validation study in Section 4.4, quality changes will be computed on isolated, executable patches

that pass the specified checks. For lower-is-better metrics, relative change is calculated as:

$$\text{Improvement} = ((\text{Before} - \text{After}) / \text{Before}) \times 100. \quad (10)$$

For higher-is-better metrics such as Maintainability Index, the formula is:

$$\text{Improvement} = ((\text{After} - \text{Before}) / \text{Before}) \times 100. \quad (11)$$

3.5 Candidate transformation validation protocol

This protocol specifies a prospective validation stage, not a completed experiment. For each held-out recommendation, record the original repository and commit, component identity, predicted operation and score, and required transformation arguments. A type such as Extract Method alone does not identify the statement range or new signature. A supported refactoring tool or a documented developer procedure must instantiate these arguments independently of the held-out target label; record manual assistance and unsupported cases separately.

First, reconstruct and build the unmodified revision under a pinned JDK and dependency environment, and run its baseline tests. Record pre-existing failures and exclude non-executable baselines only under a predefined rule, retaining their counts and reasons. Next, check operation-specific preconditions: scope and variable binding for extraction, receiver access and dispatch for movement, and visibility, overriding, and interface compatibility for hierarchy changes. A failed precondition rejects the candidate rather than silently changing the recommendation.

Apply only the recommended transformation in an isolated working copy. Preserve the patch and tool log, rebuild, and rerun the unchanged baseline regression tests. Where feasible, add characterization or differential tests that compare observable outputs, exceptions, and state changes on identical inputs. Record test coverage and any untested changed code. Static analysis provides

supplementary evidence; neither static checks nor successful compilation alone constitutes behavior validation.

Assign each case one terminal status: not executable at baseline, unsupported or underspecified, precondition rejected, application failed, build failed, behavioral discrepancy detected, checks passed, or inconclusive. A checks-passed case indicates no detected regression under the recorded checks, not a proof of semantic equivalence. Compute quality deltas only after this gate and preserve all rejected and inconclusive cases in the outcome manifest.

4. Experimental setup

4.1 Research questions

RQ1 asks how well the model predicts the recorded refactoring type. RQ2 asks whether metric and relation components add predictive value under matched settings. RQ3 compares the model with specified baselines. RQ4 asks whether feature attributions provide interpretable evidence, without presuming developer usefulness. RQ5 asks what structural-quality changes are observed, distinguishing historical, estimated, and validated transformations. RQ6 asks what proportion of held-out recommendations can be instantiated and pass the behavior-preservation checks in Section 4.4. RQ6 remains unanswered by the current experiments.

4.2 Baselines, tools, and training configuration

RABERT-RR was compared with metric-only classical models, token-based deep learning models, transformer-based code models, and an LLM-based prompt baseline. The same train-validation-test split was used for comparable models. Classical machine learning models were implemented with Scikit-learn [31], gradient boosting with XGBoost [32], and transformer models with PyTorch [33] and Hugging Face Transformers [34]. AdamW was used for transformer fine-tuning [30].

Table 4. Baseline models and input features.

Category	Baseline models	Input features
Classical machine learning	Decision Tree; Logistic Regression; SVM; Random Forest; XGBoost	Software metrics
Deep learning	LSTM; BiLSTM	Code tokens
Transformer-based	CodeBERT; GraphCodeBERT	Source-code tokens
LLM-based baseline	Prompt-based recommendation	Code component and smell label
Proposed model	RABERT-RR	Source-code embeddings; software metrics; metric relationships

Table 5. Implementation environment and training configuration.

Item	Value
Programming and ML tools	Python; PyTorch; Hugging Face Transformers; Scikit-learn; XGBoost; Pandas; NumPy; Matplotlib
Explainability tools	SHAP; LIME
Refactoring mining	RefactoringMiner
Learning rate	2e-5
Batch size	16
Epochs	20
Optimizer	AdamW
Dropout	0.3
Maximum sequence length	512
Loss function	Categorical cross-entropy
Early stopping	Validation F1-score

4.3 Evaluation metrics and statistical analysis

The multi-class recommendation task was evaluated using accuracy, precision, recall, F1-score, AUC, macro-average, weighted-average, and confusion-pattern analysis. AUC was computed using one-vs-rest macro averaging. Quality-impact metrics included Maintainability Index, cyclomatic complexity, Coupling Between Objects, Lack of Cohesion in Methods, LOC, method count, code smell count, and technical debt indicator. Statistical significance was evaluated with Wilcoxon signed-rank tests when normality could not be assumed [35]. For broader multi-model comparisons, Friedman and Nemenyi procedures are appropriate for repeated classifier comparisons [36]. Threats to validity were organized following standard empirical software engineering principles [37], [38].

4.4 Planned executable validation study

The study will use the fixed held-out test partition, without selecting cases based on favorable outcomes. If validating all 1,661 reported test instances is infeasible, freeze a stratified sample across refactoring types and repositories before execution, publish its identifiers and seed, and state the sampling rule. Do not invent a sample size or treat the 1,661 classification instances as already validated patches.

Let N be all selected recommendations, E the cases with executable baselines, A the successfully applied and built patches among E , and V the patches passing all specified behavioral checks among A . Report E/N as baseline executability, A/E as application-and-build yield, V/A as conditional check-pass rate, and V/N as end-to-end validated yield, together with counts for every

terminal status. Use “not estimable” for a zero denominator. Report per-type and per-repository outcomes with uncertainty intervals; account for within-repository dependence when estimating pooled uncertainty.

For validated patches, recompute before/after metrics using identical tools, versions, configurations, and aggregation boundaries. Extraction must include both source and newly created components so that merely moving code does not appear as deletion. Report cohort size, absolute paired deltas, relative changes, and uncertainty; relative change is undefined when the baseline is zero. Analyze historical commits and structural estimates in separate cohorts, and do not attribute changes from mixed-purpose commits to the model.

No N, E, A, or V values, regression outcomes, coverage statistics, or validated quality deltas are available in the supplied manuscript. These outcomes must be populated from actual executions before the behavior-preservation concern can be considered addressed empirically.

4.5 Replication package and release verification

The release must contain the processed 11,072-instance dataset or a legally redistributable reconstruction path; a data dictionary with units, missing-value rules, label definitions and class counts; repository URLs and licenses; parent/child commit hashes; component paths and stable sample identifiers; mined operation records; filtering and manual-adjudication decisions; and exact training, validation, and test membership. Document exclusions and redistribution restrictions at file level.

Provide runnable mining, preprocessing, training, inference, explanation, and evaluation entry points; pinned dependency and JDK versions; hardware specifications; all seeds and configurations; checkpoint identifiers or weights where permitted; and machine-readable predictions and logs for every reported model. Record the exact LLM model/version, prompt, decoding settings, query date, response parsing, failures, and class-score derivation for AUC. A

predicted class label alone is insufficient to reproduce a score-based AUC.

Document normalization fitted on training data only, duplicate and near-duplicate handling, and repository/commit overlap between splits. Features must come from the pre-change revision, with post-change information excluded from model inputs. Record the smell label’s actual role: Equation (8) conditions on S, but the presented fusion equations do not specify its encoding. Supply the implemented metric-token shapes, relation construction, attention dimensions, fusion operation, and checkpoint-selection rule instead of relying only on schematic equations.

Include each validation patch, precondition decision, build/test command, baseline and transformed logs, coverage evidence, terminal status, and paired metric record. Statistical scripts must regenerate per-class and aggregate metrics, uncertainty intervals, and tests from retained predictions and paired run records. Specify averaging conventions, independent units, effect-size definitions, and correction for multiple comparisons.

Before release, run the documented workflow in a clean environment, regenerate the manuscript tables and figures, verify split counts and output checksums, and record any deviations. Archive an immutable version with a persistent identifier, licenses, a README, expected outputs, and a release manifest. Verify anonymous access to the processed data and package; do not substitute an empty repository or an availability-on-request statement for the complete public release. Release completion remains pending.

5. Results and discussion

5.1 Overall performance

Table 6 preserves the originally reported performance values. RABERT-RR has the highest reported scores, including accuracy 0.913 and F1-score 0.902, compared with GraphCodeBERT F1-score 0.861. However, the aggregate metrics are inconsistent with Table 7, as detailed below. These values should not support definitive superiority claims until prediction-level recomputation is complete. The LLM comparison additionally

requires its exact model identifier, prompt, inference configuration, and scoring procedure.

Table 6. Performance comparison of refactoring recommendation models. Original reported values pending reconciliation or audit.

Model	Accuracy	Precision	Recall	F1-score	AUC
Decision Tree	0.742	0.731	0.724	0.727	0.781
Logistic Regression	0.758	0.746	0.739	0.742	0.796
SVM	0.771	0.762	0.754	0.758	0.811
Random Forest	0.803	0.794	0.787	0.790	0.842
XGBoost	0.821	0.813	0.806	0.809	0.861
LSTM	0.794	0.782	0.775	0.778	0.832
BiLSTM	0.812	0.804	0.796	0.800	0.851
CodeBERT	0.856	0.849	0.841	0.845	0.891
GraphCodeBERT	0.872	0.864	0.858	0.861	0.907
LLM-based baseline	0.834	0.821	0.815	0.818	0.866
RABERT-RR	0.913	0.906	0.899	0.902	0.941

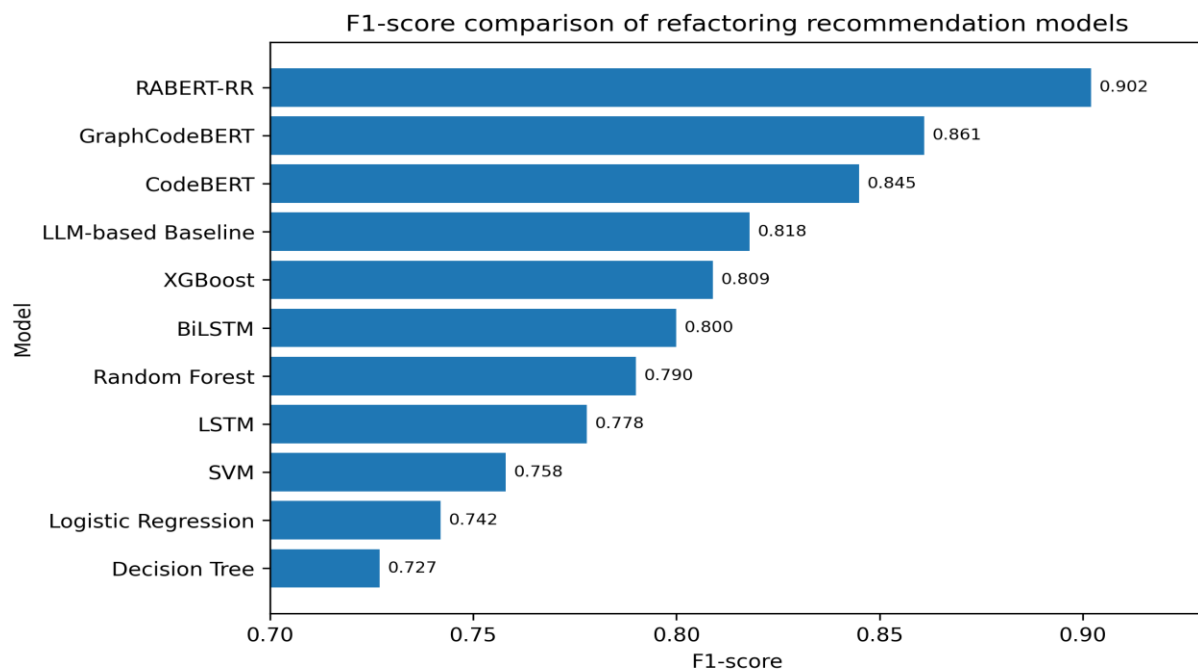


Fig. 3. F1-score comparison of RABERT-RR with baseline models. Data visualization generated from the experimental results. Underlying values remain subject to the result audit.

5.2 Performance by refactoring type

Table 7 reports higher per-class F1-scores for Extract Method, Rename Method, and Extract

Class than for inheritance-related operations. This is a descriptive pattern; the experiment does not isolate frequency, hierarchy reasoning, or

behavior-preservation constraints as its cause. The per-class supports total 1,661, but their weighted

aggregates do not match the table's weighted-average row.

Table 7. Performance by refactoring type on the test set. Original reported values pending reconciliation or audit.

Refactoring type	Precision	Recall	F1-score	Support
Extract Method	0.921	0.914	0.917	276
Rename Method	0.918	0.907	0.912	256
Extract Class	0.906	0.896	0.901	229
Move Method	0.894	0.884	0.889	182
Encapsulate Field	0.887	0.876	0.881	164
Decompose Conditional	0.874	0.861	0.867	142
Extract Interface	0.862	0.848	0.855	122
Pull Up Method	0.846	0.831	0.838	104
Push Down Method	0.832	0.817	0.824	96
Replace Conditional with Polymorphism	0.821	0.803	0.812	90
Macro average	0.876	0.864	0.870	1,661
Weighted average	0.906	0.899	0.902	1,661

Result reconciliation required: weighting the displayed class precision, recall, and F1-scores by their supports yields approximately 0.888, 0.877, and 0.882, respectively, rather than 0.906, 0.899, and 0.902. In exhaustive single-label classification, support-weighted recall also equals accuracy, so

0.899 recall and 0.913 accuracy cannot describe the same predictions under that convention. These calculations use rounded table entries and are diagnostics, not replacement experimental results. Recompute both tables from saved predictions.

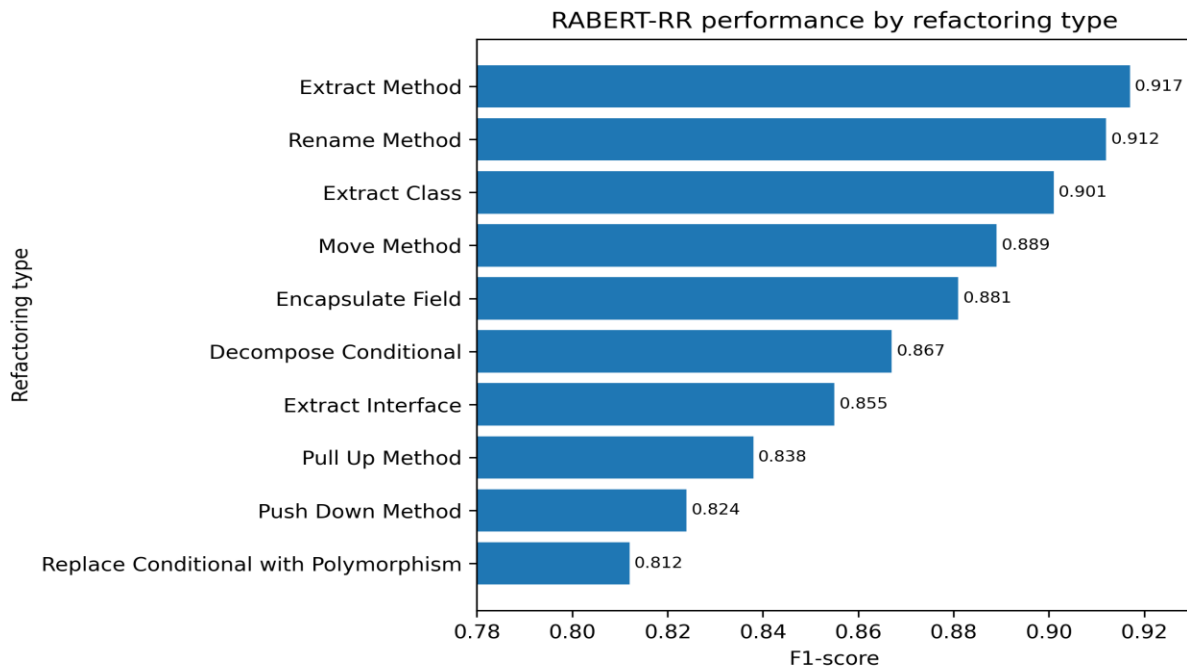


Fig. 4. F1-score of RABERT-RR across refactoring types. Data visualization generated from the experimental results. Underlying values remain subject to the result audit.

5.3 Ablation study and statistical significance

Table 8 retains the reported ablation values, which suggest an association between adding metrics, fusion, and higher F1-score. A causal attribution to relation modeling requires matched initialization, splits, tuning budgets, and repeated-

run predictions. The CodeBERT-only score also differs between Table 6 (0.845) and Table 8 (0.852); the corresponding configurations and run identities must be documented before interpreting that difference.

Table 8. Ablation study of RABERT-RR components. Original reported values pending reconciliation or audit.

Model variant	Accuracy	Precision	Recall	F1-score
Metrics only	0.842	0.836	0.831	0.833
Code tokens only	0.781	0.773	0.768	0.770
CodeBERT embeddings only	0.862	0.855	0.849	0.852
CodeBERT + metrics	0.884	0.877	0.871	0.874
CodeBERT + metrics + fusion layer	0.896	0.889	0.883	0.886
Full RABERT-RR model	0.913	0.906	0.899	0.902

Table 9 retains the originally reported p-values and effect sizes as unverified statistical summaries. The manuscript does not provide the paired observations, number of runs, seeds, effect-size definition, or multiple-comparison treatment

needed to audit these tests. Statistical significance and practical significance are therefore not established in this revision. Reanalysis must use documented independent units and report uncertainty alongside effect sizes.

Table 9. Statistical significance testing results. Original reported values pending reconciliation or audit.

Comparison	Test	p-value	Effect size	Result
RABERT-RR vs CodeBERT	Wilcoxon signed-rank	0.018	0.62	Unverified
RABERT-RR vs GraphCodeBERT	Wilcoxon signed-rank	0.041	0.49	Unverified
RABERT-RR vs XGBoost	Wilcoxon signed-rank	0.009	0.71	Unverified
RABERT-RR vs LLM baseline	Wilcoxon signed-rank	0.022	0.58	Unverified

5.4 Explainability and quality impact

Table 10 gives qualitative examples of feature-based interpretations consistent with the stated design rationale. These examples do not establish attribution fidelity, stability, or developer usefulness. A reproducible explanation analysis

must specify the explained output, SHAP background data and explainer, LIME perturbation scheme and sample budget, random seeds, and whether metrics are perturbed while code is held fixed. Correlated or implausible perturbations can affect interpretation.

Table 10. Example explainability output.

Detected smell	Recommendation	Influential features	Interpretation
Large Class	Extract Class	LOC; method count; attribute count; LCOM; CBO	The class shows excessive responsibility and low modularity.
Long Method	Extract Method	Method length; cyclomatic complexity; nesting depth; Halstead difficulty	The method contains long and complex logic that can be decomposed.
Feature Envy	Move Method	External attribute access; coupling; low local data usage	The method uses more data from another class than from its own class.
Data Class	Encapsulate Field	Public attributes; low behavior count; weak encapsulation	The class exposes data directly and requires stronger encapsulation.
Complex Conditional	Replace Conditional with Polymorphism	Conditional depth; repeated branching; type checking	Conditional behavior may be represented through polymorphic classes.

Table 11 preserves the original exploratory quality summary for traceability. Its sample count,

aggregation unit, before/after pairing, metric-tool configuration, and distinction between measured

and estimated values are not documented. Consequently, these figures cannot establish that RABERT-RR caused quality improvements or preserved behavior. They must be replaced with

separately reported historical, estimated, and validation-passing cohorts before being used as empirical evidence.

Table 11. Original exploratory quality summary pending provenance and validation.

Quality metric	Before	After	Absolute change	Percentage change
Maintainability Index	62.40	71.85	+9.45	+15.14%
Cyclomatic Complexity	18.72	12.36	-6.36	-33.97%
Coupling Between Objects	14.80	10.25	-4.55	-30.74%
Lack of Cohesion in Methods	0.68	0.49	-0.19	-27.94%
LOC	286.50	219.30	-67.20	-23.46%
Method Count	18.40	14.10	-4.30	-23.37%
Code Smell Count	1.00	0.32	-0.68	-68.00%
Technical Debt Indicator	124.60 min	81.25 min	-43.35 min	-34.79%

Note. Negative percentage values indicate reduction for lower-is-better metrics.

5.5 Discussion of research questions

RQ1: The reported accuracy is 0.913; consistency checks prevent a definitive interpretation until predictions and aggregate metrics are reconciled.

RQ2: Reported ablations favor the full model, but matched configurations and repeated-run evidence are needed to quantify the contribution of relation modeling.

RQ3: The reported ranking favors RABERT-RR. Reproducible baseline specifications and reconciled results are needed before claiming robust superiority.

RQ4: Qualitative attribution examples are provided; explanation fidelity, stability, and developer usefulness are not established.

RQ5: Exploratory quality summaries are available, but mixed provenance prevents causal or validated quality claims. RQ6: No executable-patch validation results are available; applicability and behavior-preservation outcomes remain to be measured.

6. Threats to validity

6.1 Internal and construct validity

Internal validity may be affected by noisy refactoring labels. Repository-mined refactorings are sometimes mixed with bug fixes, feature additions, formatting changes, or architectural updates. To reduce this threat, duplicate records, incomplete entries, and ambiguous low-confidence instances were removed or manually reviewed. Nevertheless, label noise remains an important limitation because supervised models learn from the quality of the mined labels.

The smell-refactoring mapping may also introduce bias. A single smell can often be resolved by several valid refactorings. Treating the task as single-label classification simplifies evaluation but does not fully capture real maintenance scenarios. Future work should extend the framework to multi-label and sequence-based recommendation.

Construct validity requires distinguishing a mined label, a predicted operation type, an executable patch, and evidence about behavior. Agreement with a mined label does not show that a recommended patch is applicable or behavior

preserving. SHAP/LIME attributions and improved structural metrics do not establish semantic equivalence. The proposed validation protocol reduces this gap only when it is executed and its results are reported.

6.2 External and conclusion validity

External validity is limited by the use of Java-based object-oriented systems. Java has strong tooling support for refactoring mining and metric extraction, but the findings may not generalize directly to Python, C++, C#, JavaScript, Kotlin, or TypeScript. Repository-domain diversity may also influence generalizability if the selected projects are concentrated in specific domains such as libraries, frameworks, or developer tools.

Conclusion validity is limited by class imbalance, unknown run-level variation, inconsistent aggregate metrics, and incomplete statistical provenance. The existing p-values cannot resolve these concerns without paired observations and a documented testing procedure. Project overlap or near-duplicate components across partitions could inflate performance; the current split description does not establish repository-disjoint evaluation. Behavior preservation remains unvalidated. Section 3.5 defines an operational validation gate, and Section 4.4 specifies how to measure its outcomes. Neither passing tests nor static checks proves equivalence for all executions. Missing tests, environmental failures, unsupported transformations, and manual choices must be reported explicitly. Until these experiments are completed, RABERT-RR is a candidate-type recommender rather than an automated refactoring engine.

7. Conclusion and future work

This paper presents RABERT-RR as an explainable candidate refactoring-type recommender combining CodeBERT representations with software metrics and inter-metric relationships. Its explanations expose influential features; they are not correctness certificates.

The manuscript reports an experiment on 11,072 Java instances, including accuracy 0.913 and F1-

score 0.902. Inconsistencies between aggregate and per-class metrics require correction from saved predictions. Historical and estimated quality summaries cannot establish behavior-preserving improvement. Definitive effectiveness claims therefore await result reconciliation, executable-patch evaluation, and release of the processed dataset and complete replication package.

The immediate next study will execute the validation protocol, separate measured outcomes from estimates, and archive the artifacts specified in Section 4.5. Subsequent work can extend to repository-disjoint and cross-language evaluation, multiple valid labels, refactoring sequences, richer structural representations, and developer studies. These extensions should follow completion of the core validity and reproducibility requirements.

Declarations

CRedit authorship contribution statement

Israr Ali: Conceptualization, methodology, software, data curation, formal analysis, investigation, visualization, writing - original draft. Ali Ahmed Siddiqui: Supervision, validation, formal analysis, project administration, writing - review and editing. Aarij Mahmood Hussaan: Software, data curation, validation, visualization, writing - review and editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Data availability

A public release of the processed dataset and complete replication package is not established in this manuscript. No permanent repository URL, archive identifier, or release version has been provided. Section 4.5 specifies the required contents and verification procedure. Before

resubmission, the authors must archive the release and replace this statement with its verified persistent identifier, version, access conditions, and licenses. This revision does not claim that the artifacts are publicly available.

Declaration of generative AI and AI-assisted technologies in the manuscript preparation process

During the preparation of this work, the authors used ChatGPT by OpenAI to support language editing, formatting according to journal guidelines, and preparation of submission-ready statements. The authors reviewed and edited the content as needed and take full responsibility for the content of the submitted manuscript. AI-assisted tools were also used to support conceptual schematic preparation for Figures 1 and 2; the figures were reviewed by the authors for accuracy and originality.

Acknowledgements

The authors thank the maintainers of the open-source repositories used for empirical evaluation and the software engineering research community for tools and datasets that support reproducible refactoring research.

REFERENCES

- [1] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, Dept. Comput. Sci., Univ. Illinois at Urbana-Champaign, Urbana, IL, USA, 1992.
- [2] M. Fowler, Refactoring: Improving the Design of Existing Code, 2nd ed. Boston, MA, USA: Addison-Wesley, 2018.
- [3] T. Mens and T. Tourwé, "A survey of software refactoring," IEEE Transactions on Software Engineering, vol. 30, no. 2, pp. 126-139, Feb. 2004.
- [4] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code. Boston, MA, USA: Addison-Wesley, 1999.
- [5] G. Lacerda, F. Petrillo, M. Pimenta, and Y.-G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," Journal of Systems and Software, vol. 167, Art. no. 110610, 2020.
- [6] N. Moha, Y.-G. Guéhéneuc, 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, vol. 36, no. 1, pp. 20-36, Jan.-Feb. 2010.
- [7] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia, "Detecting bad smells in source code using change history information," in Proc. 28th IEEE/ACM Int. Conf. Automated Software Engineering (ASE), 2013, pp. 268-278.
- [8] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," IEEE Transactions on Software Engineering, vol. 43, no. 11, pp. 1063-1088, Nov. 2017.
- [9] A. A. B. Baqais and M. Alshayeb, "Automatic software refactoring: A systematic literature review," Software Quality Journal, vol. 28, no. 2, pp. 459-502, 2020.
- [10] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: A multi-objective approach," Automated Software Engineering, vol. 20, no. 1, pp. 47-79, 2013.
- [11] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in Proc. Findings of the Association for Computational Linguistics: EMNLP, 2020, pp. 1536-1547.
- [12] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "GraphCodeBERT: Pre-training code representations with data flow," in Proc. Int. Conf. Learning Representations (ICLR), 2021.
- [13] E. Murphy-Hill and A. P. Black, "Refactoring tools: Fitness for purpose," IEEE Software, vol. 25, no. 5, pp. 38-44, Sep.-Oct. 2008.

- [14] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in Proc. ACM SIGSOFT 20th Int. Symp. Foundations of Software Engineering (FSE), 2012, pp. 1-11.
- [15] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in Proc. 31st Int. Conf. Neural Information Processing Systems, Long Beach, CA, USA, 2017, pp. 4768-4777.
- [16] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why should I trust you? Explaining the predictions of any classifier," in Proc. 22nd ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining, San Francisco, CA, USA, 2016, pp. 1135-1144.
- [17] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, and A. Chávez, "Does refactoring improve software structural quality? A longitudinal study of 25 projects," in Proc. XXX Brazilian Symp. Software Engineering, 2016, pp. 73-82.
- [18] S. Kaur and P. Singh, "How does object-oriented code refactoring influence software quality? Research landscape and challenges," *Journal of Systems and Software*, vol. 157, Art. no. 110394, 2019.
- [19] A. A. B. Baqais and M. Alshayeb, "Automatic software refactoring: A systematic literature review," *Software Quality Journal*, vol. 28, no. 2, pp. 459-502, 2020.
- [20] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: A multi-objective approach," *Automated Software Engineering*, vol. 20, no. 1, pp. 47-79, 2013.
- [21] N. Tsantalis, A. Ketkar, and D. Dig, "RefactoringMiner 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930-950, Mar. 2022, doi: 10.1109/TSE.2020.3007722.
- [22] D. Silva, J. P. da Silva, G. Santos, R. Terra, and M. T. Valente, "RefDiff 2.0: A multi-language refactoring detection tool," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2786-2802, Dec. 2021, doi: 10.1109/TSE.2020.2968072.
- [23] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? Confessions of GitHub contributors," in Proc. 24th ACM SIGSOFT Int. Symp. Foundations of Software Engineering, Seattle, WA, USA, 2016, pp. 858-870.
- [24] E. A. AlOmar, A. Ouni, M. W. Mkaouer, and M. Kessentini, "On preserving the behavior in software refactoring: A systematic mapping study," *Information and Software Technology*, vol. 140, Art. no. 106675, 2021.
- [25] B. Nyirongo, Y. Jiang, H. Jiang, and H. Liu, "A survey of deep learning based software refactoring," *arXiv preprint arXiv:2404.19226*, 2024.
- [26] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2021, pp. 2655-2668.
- [27] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. C. S. Lui, and H. Wang, "Large language models for software engineering: Survey and open problems," *ACM Transactions on Software Engineering and Methodology*, 2024.
- [28] S. Martinez, L. Xu, and M. Elnaggar, "Software refactoring research with large language models: A systematic literature review," *Journal of Systems and Software*, vol. 235, Art. no. 112762, 2025.
- [29] J. Shrivastava, D. Larochelle, and D. Tarlow, "Repository-level code generation using LLMs and planning," *arXiv preprint arXiv:2309.12499*, 2023.
- [30] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in Proc. Int. Conf. Learning Representations, 2019.
- [31] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825-2830, 2011.

- [32] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in Proc. 22nd ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining, San Francisco, CA, USA, 2016, pp. 785-794.
- [33] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," in Proc. Advances in Neural Information Processing Systems, vol. 32, 2019.
- [34] T. Wolf et al., "Transformers: State-of-the-art natural language processing," in Proc. 2020 Conf. Empirical Methods in Natural Language Processing: System Demonstrations, 2020, pp. 38-45.
- [35] F. Wilcoxon, "Individual comparisons by ranking methods," Biometrics Bulletin, vol. 1, no. 6, pp. 80-83, 1945.
- [36] J. Demšar, "Statistical comparisons of classifiers over multiple data sets," Journal of Machine Learning Research, vol. 7, pp. 1-30, 2006.
- [37] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, Experimentation in Software Engineering. Berlin, Germany: Springer, 2012.
- [38] W. R. Shadish, T. D. Cook, and D. T. Campbell, Experimental and Quasi-Experimental Designs for Generalized Causal Inference. Boston, MA, USA: Houghton Mifflin, 2002.
- [39] Ali, Israr, Syed Sajjad Hussain Rizvi, and Syed Hasan Adil. "Enhancing software quality with AI: A transformer-based approach for code smell detection." *Applied Sciences* 15.8 (2025): 4559.
- [40] Ali, I., et al. "Code smell detection and software refactoring research: A systematic literature review." *The Asian Bulletin of Big Data Management* 4.1 (2024): 107-119.
- [41] Ali, Israr, Ali Ahmed Siddiqui, and Aarij Mahmood Husaan. "A Comparative Review of Transformer Architectures: Evolution, Efficiency, Applications, and Future Directions." *Iqra Journal of Engineering and Computing* 2.1 (2026).
- [42] Ali, Israr. "A Comparative Analysis of Code Smell Detection: Rule-Based, Machine Learning, and Deep Learning Approaches." *Journal of Computers and Intelligent Systems* 3.4 (2025): 256-270.