

Integrating Structural and Semantic Analysis for Code Smell Refactoring Prediction

Andrea Lanza

University of Milano-Bicocca

Matteo Bochicchio

`matteo.bochicchio@unimib.it`

University of Milano-Bicocca

Francesca Arcelli Fontana

University of Milano-Bicocca

Research Article

Keywords: Code Smells, Technical Debt, Refactoring, Machine Learning, Predictive Models

Posted Date: October 16th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-11220560/v1>

License:   This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

Integrating Structural and Semantic Analysis for Code Smell Refactoring Prediction

Andrea Lanza, Matteo Bochicchio, Francesca Arcelli Fontana
Department of Informatics, Systems and Communication (DISCo),
University of Milano-Bicocca, Milan, Italy.

*Corresponding author(s). E-mail(s): matteo.bochicchio@unimib.it;
Contributing authors: a.lanza13@campus.unimib.it;
francesca.arcelli@unimib.it;

Abstract

Code smells are localized flaws that hinder software maintainability, but their presence does not always prompt developers to refactor. While existing static analysis tools provide accurate detection of code smells and compute traditional Object-Oriented metrics, often used to detect the smells, such metrics fundamentally abstract away the sequential logic and behavioural flow of the code. To capture these semantic and structural properties, source code can be modelled as graphs and mapped into a continuous latent vector space, which is then fed into machine learning models to predict refactoring actions. To achieve this goal, this paper proposes the design of a machine learning framework applied to Java projects, focusing on the empirical evaluation of predictive models that integrate traditional metrics with semantic and syntactically rich embeddings. The results of this empirical study show that machine learning models are reliable predictors; and while traditional metrics account for the vast majority of feature importance when paired with Control Flow Graph-derived embeddings, transitioning to a combination of both semantic and syntactic representations introduces a complementary, and independent signal.

Keywords: Code Smells, Technical Debt, Refactoring, Machine Learning, Predictive Models

1 Introduction

Software evolution and maintenance drive the lifecycle of modern systems; however, continuously introducing changes often leads to internal structure degradation, making future software changes progressively more costly and difficult to implement, a phenomenon commonly referred to as Technical Debt (TD) [1].

At the implementation level, TD manifests as code smells [2, 3]. To resolve these flaws and improve readability and maintainability without altering the observable external behavior of the system, developers employ refactoring [2]. Traditionally, developers rely on static analysis tools to compute Object-Oriented (OO) metrics [4]. However, traditional metrics abstract away the sequential logic of the program. By evaluating a method as a static collection of properties rather than its execution path, these metrics fail to track how data and control propagate, lacking an understanding of semantic and behavioral context [5]. Predicting refactoring operations allows developers to forecast which flaws genuinely warrant intervention based on past history, shifting from passive detection toward predictive TD management [6].

Recent advancements have thus explored Machine Learning (ML) models as predictors of refactoring operations [7, 8], and to overcome the limitations of standard metrics and capture program semantics, and data/control flow dependencies, ML architectures can be fueled by graph-based code representations in the form of continuous vector embeddings.

In our study, we extract historical refactoring data from open-source Java repositories utilizing the RefactoringMiner tool [9]. We engineer a hybrid feature space combining traditional OO metrics extracted via Designite [10] and CK Analysis Tool [11] with:

1. Control Flow Graph (CFG) vector embeddings generated using Joern [12] and the LINE algorithm [13];
2. Joint semantic and syntactic vector embeddings generated via GraphCode2Vec [14] (integrating Jimple IR, Program Dependence Graph (PDG), and Bidirectional Long Short-Term Memory (BiLSTM)). For a complete description of the framework, see Section 2.4.

We formulate the task as a binary classification problem to evaluate predictive capabilities (**RQ1**) and analyze feature importance (**RQ2**), measured as the relative contribution of each feature set to the model’s decisions. We provide an empirical evaluation benchmarking four supervised ML classifiers across 10 method- and variable-level refactoring operations on 79 Java projects. Our results show that supervised ML models are effective predictors of code smell refactoring, with F1-scores frequently surpassing 0.70 and peaking above 0.80 (RQ1). Furthermore, ablation and interpretability analyses reveal a contrast between code representations (RQ2): while CFG embeddings introduce feature noise and are largely ignored in favor of OO metrics ($\approx 85\%$ feature importance share), multi-modal GraphCode2Vec embeddings provide complementary signals that actively drive predictions ($\approx 60\%$ feature importance share), allowing stand-alone embedding models to achieve competitive performance.

The complete data and scripts used for this study are available in the publicly accessible online replication package [15].

The remainder of this paper is organized as follows: Section 2 provides foundational background; Section 3 outlines the study design and experimental setup; Section 4 presents the empirical results; Section 5 discusses lessons learned from the results; Section 6 discusses the limitations of the study; Section 7 reviews related work and Section 8 concludes the paper.

2 Background

2.1 Code Smells and Designite

Code smells occur at the granular level of methods, statements, and variables, representing low level problems localized within the source code itself [2].

In this study the identification of code smells is automated through the use of **Designite** tool [10]. Designite is a state-of-the-art Java code quality analysis tool that computes a suite of 14 software metrics and supports the detection of a comprehensive set of code smells across various abstraction levels. The tool is able to identify architectural, design, and implementation smells, the latter commonly referred to in the literature as code smells. Designite has been widely used in several empirical studies [16–19].

The following is the list of code smells detected by Designite and considered in our study:

- *Abstract Function Call From Constructor*: An OO initialization flaw where a constructor invokes an abstract or overridable method [20].
- *Complex Conditional*: This smell occurs when a conditional statement is complex [20].
- *Complex Method*: This smell occurs when a method has high cyclomatic complexity [20].
- *Empty Catch Clause*: This smell occurs when a catch block of an exception is empty [20].
- *Long Identifier*: This smell occurs when an identifier is excessively lengthy [20].
- *Long Method*: This smell occurs when a method is too long to understand [2].
- *Long Statement*: This smell occurs when a statement is excessively lengthy [20].
- *Long Parameter List*: This smell occurs when a method accepts a long list of parameters [2].
- *Magic Number*: This smell occurs when an unexplained number is used in an expression [20].
- *Missing Default*: This smell occurs when a switch statement does not contain a default case [20].

2.2 Refactorings

Martin Fowler [2] defines refactoring as a technique for restructuring an existing body of code, to improve readability, maintainability, and extensibility without changing its observable behavior. To collect a sufficient volume of labeled data (refactorings) for empirical studies on software evolution, an automated mining tool is required, such as **RefactoringMiner** serving as the de facto standard. RefactoringMiner [9] is an

advanced static analysis framework engineered to automatically detect and classify refactoring operations executed across the commit history of Git repositories. RefactoringMiner has been widely adopted in the literature due to its performance and its independence from external build configurations [8, 21–23]; it operates directly on raw source code streams without requiring successful compilation of the projects.

For our study we have considered the following refactorings based on Fowler’s definitions [2]: *Extract Method*, *Move Method*, *Rename Method*, *Extract Variable*, *Rename Variable*, *Rename Parameter*, *Add Parameter*, *Change Variable Type*, *Change Parameter Type*, *Change Return Type*.

2.3 Object-Oriented Metrics

The theoretical foundation of OO measurement relies heavily on the metric suite proposed by Chidamber and Kemerer (CK suite) [24].

The CK framework formally defines six distinct metrics targeting the structural dimensions of an OO project: *Weighted Methods per Class (WMC)*, *Depth of Inheritance Tree (DIT)*, *Number of Children (NOC)*, *Coupling Between Objects (CBO)*, *Response for a Class (RFC)* and *Lack of Cohesion of Methods (LCOM)*. To compute these metrics in practice on large-scale projects, the adoption of automated extraction tools becomes essential; in this context, Designite and the CK Analysis Tool are widely employed to collect metric suites across different abstraction levels. The CK Analysis Tool [11] is an open-source tool designed to compute OO metrics by operating directly on the source code’s Abstract Syntax Tree (AST) via the Eclipse Java Development Tools (JDT) compiler framework. Together, both tools provide a comprehensive evaluation of software artifacts: Designite extracts structural metrics alongside code smell indicators, while the CK Analysis Tool calculates the standard CK metric suite as well as granular method- and class-level code features.

Table 1 provides a summary of the full metric suite extracted by both CK Tool and Designite, categorized by their level of abstraction and measurement scope.

Table 1: Extracted Software Metrics from CK Tool and Designite.

Metric	Description	Metric	Description
Designite Metrics (Class & Method)			
NOF	Number of fields in scope	NC	Number of direct subclasses
NOPF	Number of public fields	DIT	Depth of inheritance tree
NOM	Number of declared methods	LCOM	Lack of cohesion index
NOPM	Number of public methods	Fan-In	Incoming class dependencies
LQC	Lines of code (class level)	Fan-Out	Outgoing class dependencies
WMC	Weighted methods per class	method.loc	Lines of code (method body)
method.cc	McCabe cyclomatic complexity	method.pc	Method parameter count
CK Tool — Class-Level Metrics			
cbo, cboModified	Coupling between objects	fanin, fanout	Structural dependencies
wmc	Weighted methods per class	dit, noc	Inheritance tree depth & children
rfc	Response for a class	lcom, lcom*	Lack of cohesion (classic/HS)
tcc, lcc	Tight / Loose class cohesion	totalMethodsQty	Total methods defined
*MethodsQty	Counts by method modifier	totalFieldsQty	Total fields defined
*FieldsQty	Counts by field modifier	nosi	Static invocations count
loc	Lines of code aggregate	returnQty	Total return statements

Continued on next page

Table 1 (continued)

Metric	Description	Metric	Description
loopQty	Total loop constructs	comparisonsQty	Comparison operators count
tryCatchQty	Try-catch blocks count	parenthesizedExpsQty	Parenthesized expressions count
stringLiteralsQty	String literals count	numbersQty	Numerical literals count
assignmentsQty	Variable assignments count	mathOperationsQty	Math operations count
variablesQty	Declared local variables	maxNestedBlocksQty	Max nesting depth
anonymousClassesQty	Inner anonymous classes	innerClassesQty	Standard inner classes
lambdasQty	Functional lambda expressions	uniqueWordsQty	Unique source code tokens
modifiers	Access modifier bit-mask	logStatementsQty	Logging instructions count
CK Tool — Method-Level Metrics			
constructor	Constructor boolean flag	cbo, cboModified	Method-specific CBO
fanin, fanout	Method-level dependencies	wmc, rfc	Local complexity & response set
loc	Method body lines of code	returnsQty	Return statements in method
variablesQty	Local variables in method	parametersQty	Signature parameter count
methodsInvokedQty	Total external calls	methodsInvokedLocalQty	Local class method calls
methodsInvokedInd..	Indirect local calls	loopQty, comp..	Loops & comparisons count
tryCatchQty	Local try-catch blocks	parenthesizedExps..	Parenthesized expressions count
stringLit..., num..	Literal counts in method	assign..., mathOp..	Assignments & math ops
maxNestedBlocksQty	Max nesting depth in method	anonClasses..., inner..	Nested constructs & inner classes
lambdasQty, unique..	Local lambdas & unique tokens	modifiers	Method access modifier bit-mask
logStatementsQty	Logging instructions in method	hasJavaDoc	JavaDoc presence boolean flag

2.4 Topological Code Representations

While traditional metrics quantify static properties of code, they fail to capture its behavioral and execution logic. To overcome this, topological representations can be used to encapsulate control and data dependencies. We describe below the representations employed in our study:

- *Control Flow Graph (CFG)*: to capture the execution paths within a method, the code can be modeled as a CFG. In this mathematical representation, the method is defined as a directed graph, $G = (V, E)$. The vertices (V) correspond to individual statements or predicate expressions, while the directed edges (E) map the allowable transitions of control flow during execution [25].
- *Program Dependence Graph (PDG)*: unlike a CFG, which maps the chronological execution order of statements, a PDG is a directed graph where edges represent two forms of dependence: control dependence and data dependence [26].
- *Code Property Graph (CPG)*: A CPG is a topological representation that mathematically merges three distinct graph structures: the AST, CFG and PDG. By unifying syntax, control flow, and data flow into a single joint data structure, CPGs provide an enriched view of software semantics [12].

Translating source code into graphs and subsequently in embeddings requires dedicated software tools.

Joern¹ is an advanced, open-source static analysis tool capable of generating graph-based representations of source code. By parsing Java source code the tool is

¹See <https://github.com/joernio/joern>

able to construct a variety of graph representations, which provide a rich topological map of both structural and behavioral semantics of the code.

Because traditional ML algorithms require fixed in size vectors as input, graphs must be mapped into a latent space through an embedding process [27]. Another tool that we present is **LINE** (Large-scale Information Network Embedding) [13]. LINE is an algorithm explicitly designed to project large graphs into vector spaces. It achieves this by optimizing for second-order proximity, which captures the shared structural context between nodes. This ensures that methods sharing similar neighborhoods, even if not directly connected themselves, are mapped closely together in the embedding space.

GraphCode2Vec [14] is a framework which is able to simultaneously capture both sequential token syntax and structural graph topologies. It operates directly on bytecode, which is standardized through the use of Soot [28], a static analysis framework translating raw Java bytecode into Jimple (a typed intermediate representation) to normalize code structure and remove syntactic noise. Operating on this representation, GraphCode2Vec captures structural and semantic behavior via a PDG, while leveraging **Word2Vec** [29] to project individual tokens into a continuous vector space based on their semantic and operational similarity. These token streams are subsequently processed via a **BiLSTM** [30], which analyses sequences both forward and backward to capture the full contextual flow preceding and following each statement. The structural PDG dependencies and the sequential BiLSTM features are concatenated to produce a single, fixed-size continuous vector in output.

3 Study Design

This section presents the empirical methodology designed for the refactoring prediction framework, regarding research questions, data collection pipelines, feature engineering, and model training and evaluation setups.

3.1 Research Questions

We pursue two main objectives: establishing the predictive capabilities of applied ML models, and identifying the key features driving their predictions. This study is guided by two main research questions:

- **RQ1: How accurate are supervised ML algorithms in predicting code smells refactoring?** This question aims to establish the ability of supervised ML models to correctly predict specific refactoring steps using a hybrid feature space (fusion of OO metrics with topological and syntactic embeddings).
- **RQ2: Which features most significantly impact code smell refactoring prediction?** Addressing the "black-box" nature of ML [31], this question provides explainability by identifying whether OO metrics, embeddings, or their combination drive the models' decisions.

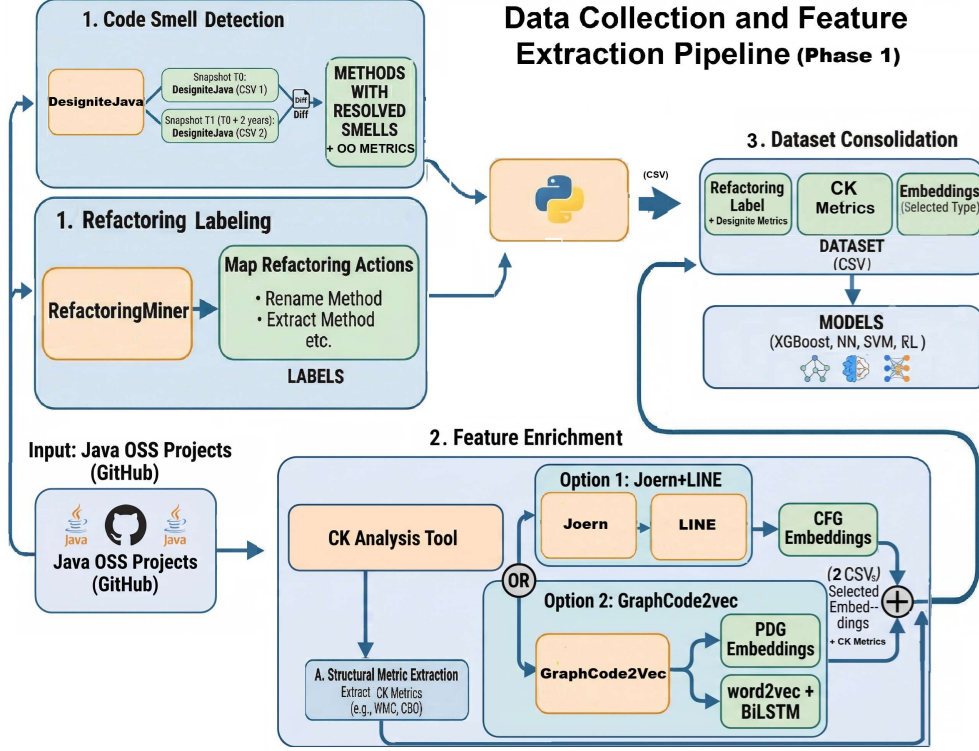


Fig. 1 Data Collection and Feature Extraction Pipeline (Phase 1). The process starts with smell detection, history collection, and refactoring labeling (our targets), followed by feature enrichment with CK metrics and embeddings, concluding with dataset consolidation.

3.2 Methodology Overview

To address these research questions, we designed an empirical study that combined traditional OO metrics with two distinct embedding strategies aimed at capturing the structural and behavioral properties of the code, across both source code and bytecode abstractions.

The methodology consists of three primary phases described below, and Figure 1 illustrates the workflow for Phase 1:

- **Phase 1 – Data Collection and Feature Extraction:** Historical refactoring data and smell-affected methods were extracted from open-source Java repositories (Step 1). Hybrid feature spaces were constructed using OO metrics combined with either a Structural (CFG via Joern+LINE) or a Joint Syntactic/Semantic (PDG + Jimple IR via GraphCode2Vec) representation (Step 2), producing a raw dataset (Step 3).

- **Phase 2 – Data Pre-processing and Feature Engineering:** Tabular data underwent cleaning, missing value handling, Z-score scaling, custom feature engineering (Smell Refactoring Ratio), and class balancing.
- **Phase 3 – Model Training, Evaluation and Interpretability:** Classifiers ranging from linear baselines to deep neural networks were optimized, trained and evaluated.

3.3 Data Collection and Feature Extraction (Phase 1)

Before the automated data collection could start, we had to select a sufficiently ample set of projects to be analyzed. We built upon the methodology established by Aniche et al. [8], who gathered a large and representative set of open-source Java projects, by selecting the 9,072 most starred projects on GitHub. We decided to apply an additional filtering step, to accommodate strict computational and temporal constraints, we ranked the projects based on volume of documented refactoring activity, and retained only the projects that maximized the density of refactoring operations until the maximum allowable computational budget and time window were saturated. This approach resulted in 79 projects, the complete list of which can be consulted in the replication package [15].

Step 1: Code Smell Detection & Refactoring Labeling.

The evolutionary history of each project was extracted using RefactoringMiner. Concurrently, a two-year window was defined, capturing project snapshots at the start and end. Designite analyzed the initial snapshot to detect 10 code smells and compute 14 metrics (See Section 2.1), while the second snapshot verified that targeted smells were resolved. A custom Python pipeline mapped filtered methods to ten target method- and variable-level refactorings [2]: Extract Method, Move Method, Rename Method, Extract Variable, Rename Variable, Rename Parameter, Add Parameter, Change Variable Type, Change Parameter Type, Change Return Type (See Table 2). There is no standard size defined in the literature for choosing the time window in which to analyze code smell refactorings. However, several studies [32, 33] highlight that such interventions are not uniformly distributed; rather, they tend to be concentrated in short strategic periods within the software lifecycle. The adopted window, therefore, aimed to provide RefactoringMiner with a temporal *delta* wide enough to allow for capturing the evolutionary history of the methods.

Step 2: Feature Enrichment.

We extracted an additional 62 OO metrics using the CK Analysis Tool (See Section 2.3). Although certain metrics overlapped between Designite and CK Analysis Tool, naming conflicts were resolved at the software level using specific prefixes, while structural redundancies were left to be managed directly by the machine learning models' intrinsic regularization algorithms.

In parallel, embeddings were generated via two alternate branches:

Table 2 Distribution of refactorings

Refactoring Operation	Count
Rename Method	1,419
Extract Method	1,405
Add Parameter	1,233
Change Return Type	1,113
Move Method	1,033
Change Variable Type	670
Change Parameter Type	341
Rename Variable	315
Extract Variable	312
Rename Parameter	308
Total	8,150

Table 3 Distribution of code smells

Code Smell	Count
Long Statement	44,233
Magic Number	25,508
Complex Method	11,454
Long Parameter List	7,956
Complex Conditional	5,451
Long Identifier	5,103
Empty catch clause	2,813
Missing default	1,011
Long Method	179
Abstr. Function Call Constr.	48
Total	103,756

- Execution Flow Pipeline (Joern + LINE): Joern extracted CFGs. LINE mapped CFGs into continuous vector spaces optimizing for second-order proximity, aggregated to method-level via mean pooling.
- Syntactic, Data and Control Flow Pipeline (GraphCode2Vec): Soot compiled byte-code into Jimple IR. GraphCode2Vec extracted PDGs for structural semantics, alongside Word2Vec+BiLSTM processing of Jimple token sequences, fusing them into multi-modal method-level embeddings.

Step 3 – Dataset Consolidation.

The previously generated artifacts were merged into a consolidated CSV (the raw dataset) containing OO metrics (Designite + CK Tool), metadata, embeddings, and binary refactoring labels. As noted in literature [22], snapshot-based evaluation across two year deltas captures evolutionary resolution without asserting strict 1-to-1 deterministic causality for individual commits.

3.4 Data Pre-processing and Feature Engineering (Phase 2)

The raw dataset required pre-processing and refinement before being utilized for model training. This phase ensured data quality, addressed class imbalance, and introduced a domain specific feature.

Initially the data was cleaned through a sanitization process, this included the removal of duplicate entries and the handling of missing values (we remove the entire row in this case) to ensure the integrity of the statistical analysis. Metadata fields such as file paths, package names, were discarded, as they provided no value to the machine learning models. To capture historical co-occurrence between smells and refactorings, we introduced a domain-specific feature, the **Smell Refactoring Ratio**. This metric quantifies the historical likelihood that a specific code smell will trigger a particular refactoring intervention. It is calculated as the frequency of a specific refactoring technique being applied in the presence of a given smell, normalized by the total occurrences of that smell within the training set. To ensure the robustness of this score and prevent overfitting on rare smells, we applied *Bayesian Smoothing* [34].

Table 4 Transposed representation of the dataset instance feature space and target vector

Feature Category	Feature Name/Label Name	Math. Domain
Method-Level	<i>method_loc, method_cc, method_parametersQty, ...</i> <i>... (54 method metrics total)</i>	$\mathbb{Z}_{\geq 0}^*$ or $\{0, 1\}$
Class-Level	<i>class_wmc, class_cbo, NOM, class_loc, ...</i> <i>... (22 class metrics total)</i>	$\mathbb{Z}_{\geq 0}^*$ or $\{0, 1\}$
Custom Engineered	<i>Smell Refactoring Ratio</i>	$\mathbb{R} \in [0, 1]$
Embeddings	<i>emb_0 ... emb_d</i>	$\mathbb{R}^d$ where $d \in \{64, 200\}$
Target Variables	<i>Extract Method, Move Method, ... (10 targets)</i>	$\{0, 1\}$

This technique mathematically blends the smell specific probability with the global average probability across the entire dataset, effectively regularizing the impact of infrequent observations. To address severe class imbalance (8,150 positive refactorings in Table 2 vs. 103,756 smell instances in Table 3), we adopted a 1:1 undersampling strategy following Aniche et al. [8], constructing balanced binary sub-datasets for both training and evaluation. Input features were normalized via Z-score Standard Scaling [35, 36] using scikit-learn [37], ensuring that features with varying magnitudes contribute equally to the learning process.

The complete feature vector combines the Smell Refactoring Ratio, 76 OO metrics, and alternatively:

- A 64-dimensional CFG embedding vector (LINE);
- A 200-dimensional semantic/syntactic embedding vector (GraphCode2Vec).

Instance structures are transposed in Table 4.

3.5 Model Training (Phase 3)

Predictive modeling was structured as ten independent binary classification tasks. We evaluated four distinct model architectures: Logistic Regression (LR), Support Vector Machines (SVM) [38], XGBoost [39], and a Feed-Forward Neural Network (NN) [40]. Datasets were partitioned using an 80/20 stratified train/test split.

Hyperparameter optimization was executed independently for each of the ten target refactoring operations:

- **XGBoost Optimization:** Randomized Search [41] optimized tree depth, learning rate, and subsample ratios, validated via 3-Fold Cross-Validation targeting macro F1-score [42].
- **Neural Network Optimization:** Topology was fixed to a two-layer MLP with ReLU activations and a sigmoid output node. Bayesian Optimization via Keras Tuner [43] tuned hidden units, dropout, L2 penalties, and learning rates. Training enforced Early Stopping [44] monitoring validation loss on a 15% holdout set.

- **Baselines:** LR and SVM were deployed using standard scikit-learn defaults (LR: LBFGS solver and max_iter=1000; SVM: RBF kernel) to establish unbiased benchmark baselines.

3.6 Evaluation and Interpretability (Phase 3)

To rigorously assess model reliability, feature contributions, and generalization capabilities, several post-hoc evaluation and diagnostic techniques were conducted.

Firstly, classification capabilities were evaluated at two granularities: Any code smell and code smell specific. Precision, Recall, F1-Score, and ROC-AUC curves were computed:

$$Precision = \frac{TP}{TP + FP}, \quad Recall = \frac{TP}{TP + FN}, \quad F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \quad (1)$$

Beyond standard performance metrics, a rigorous assessment requires investigating model behavior through explainability, feature dependency analysis, and probabilistic confidence.

Regarding visualization and model explainability, XGBoost feature importance was derived from tree split criteria (Gain/Weight). For Neural Networks, SHAP (SHapley Additive exPlanations) [45] quantified feature and macro-group (Embeddings vs. OO metrics) contributions. UMAP² projected embedding topologies onto 2D planes for visual separability inspection.

Moreover, we also conducted ablation studies. Feature choices were validated by retraining models under two restricted configurations: OO Metrics Only and Embeddings Only, isolating and testing the two feature spaces against the hybrid baseline.

Finally, regarding the probability calibration, probabilistic confidence was evaluated via Calibration Curves, Brier Score ($BS = \frac{1}{N} \sum (p_i - y_i)^2$), and Binary Cross-Entropy Log Loss [46]:

$$\mathcal{L}_{log} = -\frac{1}{N} \sum_{i=1}^N [y_i \log(p_i) + (1 - y_i) \log(1 - p_i)] \quad (2)$$

3.7 Experimental Variants

To evaluate specific feature engineering choices, five targeted variants were executed:

- **Dimensionality Reduction via PCA:** PCA compressed embedding vectors down to 3 principal components before concatenation with OO metrics for XGBoost training.
- **Stacking Ensemble (Late Fusion):** A 5-Fold OOF meta-model evaluated late-fusion combinations of XGBoost (trained on OO metrics) and Neural Network (trained on embeddings) confidence scores.
- **Use of CPG:** Evaluated CPGs generated via Joern against CFG representations.

²See <https://github.com/lmcinnes/umap>

- **Embeddings Dimensionality:** Tested LINE embedding dimensionalities across $d \in \{32, 64, 128\}$.
- **Self-Supervised Learning Paradigms:** Evaluated unsupervised autoencoding and contrastive learning pre-training strategies on the Neural Network architecture to mitigate potential data scarcity and improve representation initialization.

4 Results

This section presents the empirical evaluation of the predictive models across the 10 selected refactoring techniques, structured around our two primary research questions. The replication package containing full results and datasets is available online [15].

4.1 Answering RQ1

To address RQ1, we benchmark Neural Network and XGBoost models against traditional baselines (SVM and LR) on the hybrid feature space generated via Joern + LINE (CFG embeddings + OO metrics). The empirical results show that supervised ML models are reliable predictors of code smell refactoring. Both XGBoost and the Neural Network consistently outperform the traditional baselines, achieving robust predictive performance with F1-scores frequently surpassing the 0.70 threshold and peaking above 0.80 (Table 5). The NN achieves top performance on the hybrid CFG space for Extract Variable (F1 = 0.79), while XGBoost excels in structural transformations such as Move Method (F1 = 0.76, P = 0.76, R = 0.75) and Rename Parameter (F1 = 0.77). Linear baselines generally trail behind, though Logistic Regression serves as an outlier on Change Variable Type (F1 = 0.83).

Table 5 Predictive Performance across Refactoring Techniques (Positive Class)

Refactoring Technique	XGBoost			Neural Network			SVM			Logistic Regression		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Rename Method	0.69	0.68	0.68	0.65	0.72	0.69	0.67	0.69	0.68	0.65	0.65	0.65
Add Parameter	0.69	0.70	0.69	0.72	0.76	0.74	0.71	0.64	0.67	0.70	0.64	0.67
Extract Variable	0.73	0.76	0.75	0.72	0.87	0.79	0.78	0.74	0.76	0.70	0.82	0.76
Rename Parameter	0.85	0.71	0.77	0.84	0.68	0.75	0.88	0.58	0.70	0.82	0.73	0.77
Change Variable Type	0.83	0.78	0.80	0.78	0.79	0.79	0.81	0.83	0.82	0.81	0.85	0.83
Change Parameter Type	0.76	0.74	0.75	0.75	0.75	0.75	0.85	0.50	0.63	0.71	0.71	0.71
Change Return Type	0.68	0.69	0.68	0.67	0.69	0.68	0.65	0.60	0.62	0.62	0.61	0.61
Extract Method	0.67	0.71	0.69	0.64	0.78	0.70	0.69	0.68	0.69	0.69	0.63	0.66
Move Method	0.76	0.75	0.76	0.69	0.79	0.74	0.69	0.59	0.64	0.66	0.62	0.64
Rename Variable	0.74	0.73	0.74	0.72	0.79	0.76	0.81	0.62	0.70	0.73	0.70	0.72

Figure 2 illustrates models discrimination on the Move Method dataset. XGBoost achieves the highest Area Under the Curve (AUC), closely followed by the Deep Neural Network, while SVM and LR trail behind. Non-linear models maintain AUC values between 0.75 and 0.90 across all evaluated tasks.

To move beyond aggregate performance metrics, we grouped the predictive results across individual code smell categories. The primary objective of this granular analysis

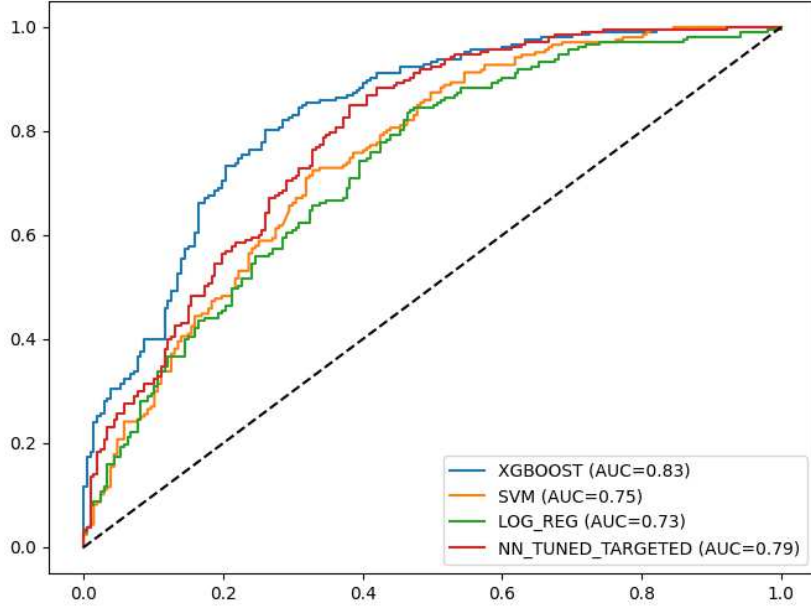


Fig. 2 ROC curves comparing the predictive performance of the 4 models on the Move Method dataset.

is to determine the models performances on specific smell-to-refactoring mappings. The predictions were derived using an ensemble model combining XGBoost and the Neural Network to leverage their joint predictive capabilities.

To ensure the integrity of this analysis, a support threshold was applied, only mappings with a proven presence of more than 15 true refactoring instances within the test set were evaluated. This successfully filters out exceedingly rare anomalies, as standard evaluation metrics such as the F1-score can become volatile and sensitive to single-instance misclassifications when too few instances are evaluated. The specific threshold was selected empirically as an optimal trade-off point based on our dataset distribution.

Table 6 highlights the most notable mappings, categorized by their predictive reliability. The strongest performing prediction within this study emerged from the mapping between the Long Parameter List smell and the Add Parameter operation (Recall = 0.94, F1 = 0.89). This high predictability aligns with standard refactoring practices; resolving Long Parameter List typically involves consolidating a scattered group of parameters into a single one, which automated refactoring trackers could register as an Add Parameter operation.

Table 6 Predictive Reliability of Specific Smell → Refactoring Mappings (Support > 15)

Code Smell	Predicted Action	F1	Prec.	Recall	True Refac.
Best Predicted Mappings (Strong Structural Catalysts)					
Long Parameter List	→ Add Parameter	0.89	0.85	0.94	64
Long Statement	→ Change Variable Type	0.87	0.87	0.87	90
Long Statement	→ Extract Variable	0.75	0.74	0.76	49
Complex Method	→ Extract Method	0.79	0.71	0.88	52
Complex Conditional	→ Extract Method	0.72	0.63	0.85	34
Worst Predicted Mappings (Weak or Noisy Signals)					
Complex Method	→ Rename Method	0.62	0.74	0.53	32
Complex Conditional	→ Rename Method	0.63	0.85	0.50	22
Magic Number	→ Extract Method	0.55	0.74	0.44	32
Magic Number	→ Rename Method	0.54	0.50	0.59	17
Magic Number	→ Change Return Type	0.47	0.78	0.33	21

On the other hand, the model precision degraded when evaluating lexical refactoring interventions. For instance, the framework failed to reliably predict a Rename Method action when faced with a Complex Method (Recall = 0.53) or a Complex Conditional smell (Recall = 0.50). From a software engineering perspective, this predictive failure is logical, simply renaming the identifier of a method does nothing to resolve its complexity or cyclomatic density, and modifying the body of a method does not always lead to a renaming. Because developers can at times rename methods independently of any refactoring, the ML algorithms could not establish a mathematical correlation between a high method complexity and a change in its identifier. The Magic Number smell consistently generated the weakest predictive signals, particularly for Extract Method, Rename Method, and Change Return Type, where the operational recall was bounded between 0.33 and 0.59.

Based on the empirical results analyzed in this section, we can formally articulate the answer to our first Research Question.

Answer to RQ1

Supervised machine learning models are reliable predictors for code smell refactoring, with F1-scores frequently surpassing 0.70 and peaking above 0.80. However, predictability strictly depends on the underlying driving factors:

- **Highly Predictable (e.g., Change Variable Type, F1 = 0.83):** Governed by localized quantifiable metrics and structural patterns captured effectively by OO metrics and graph embeddings.
- **Difficult to Predict (e.g., Rename Method, F1 = 0.69):** Intertwined with semantic intent and domain-driven conceptual updates, which are harder to capture through code representations alone.

4.2 Answering RQ2

To address RQ2, we compare the structural Joern + LINE (CFG) pipeline against the multi-modal GraphCode2Vec (bytecode PDG + Jimple token BiLSTM) paradigm via ablation and interpretability studies. The results of the ablation study detailed in Table 7 show a detrimental, or at best neutral, effect of incorporating Joern + LINE embeddings for the XGBoost model.

When measuring the ensemble-tree performance, relying exclusively on OO metrics, showed a consistent superiority. Removing the 64-dimensional graph embeddings resulted in a performance increase across the majority of refactoring techniques. For example, in predicting Move Method, the F1-score jumped from 0.76 (Full) to 0.80 (Ablated). This degradation for the XGBoost model when utilizing the full hybrid space can be theorized as a consequence of introducing underutilized data, which introduces hidden TD in machine learning frameworks [47]. The decision trees were forced to evaluate low importance structural splits, effectively obscuring the predictive signals of the traditional OO metrics.

The NN model also failed to show a strong, upward trend in overall performance; the addition of the CFG embeddings yielded fluctuating results. This architectural limitation is further illustrated by the performance of the NN Only Embeddings variant. When working with just the CFG embeddings, the model’s F1-score fell under 0.50 for Change Variable Type and Rename Variable refactoring operations.

Table 7 Ablation Comparison: CFG (Joern+LINE) Predictive Performance (Positive Class)

Refactoring Technique	XGBoost (Full)			XGBoost (Ablated)			NN (Full)			NN (Ablated)			NN Only Embeddings		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Rename Method	0.69	0.68	0.68	0.72	0.76	0.74	0.65	0.72	0.69	0.66	0.69	0.67	0.52	0.95	0.67
Add Parameter	0.69	0.70	0.69	0.76	0.74	0.75	0.72	0.76	0.74	0.68	0.73	0.70	0.52	0.74	0.62
Extract Variable	0.73	0.76	0.75	0.76	0.81	0.78	0.72	0.87	0.79	0.70	0.81	0.75	0.52	0.53	0.53
Rename Parameter	0.85	0.71	0.77	0.76	0.77	0.77	0.84	0.68	0.75	0.80	0.84	0.82	0.93	0.44	0.59
Change Variable Type	0.83	0.78	0.80	0.85	0.83	0.84	0.78	0.79	0.79	0.76	0.84	0.79	0.48	0.34	0.40
Change Parameter Type	0.76	0.74	0.75	0.82	0.78	0.80	0.75	0.75	0.75	0.75	0.72	0.74	0.64	0.60	0.62
Change Return Type	0.68	0.69	0.68	0.68	0.71	0.69	0.67	0.69	0.68	0.67	0.67	0.67	0.51	0.96	0.67
Extract Method	0.67	0.71	0.69	0.70	0.73	0.71	0.64	0.78	0.70	0.68	0.81	0.74	0.51	0.84	0.63
Move Method	0.76	0.75	0.76	0.80	0.80	0.80	0.69	0.79	0.74	0.70	0.71	0.71	0.50	1.00	0.67
Rename Variable	0.74	0.73	0.74	0.70	0.75	0.72	0.72	0.79	0.76	0.68	0.79	0.74	1.00	0.32	0.48

The performance degradation observed in the tree-based model, combined with the behavior of the NN when incorporating CFG representations, possibly imply a structural limitation in how the models process these graph embeddings. To investigate the root cause of this lack of predictive power, we analyzed the contribution of each feature set. The macro feature share calculated for the NN model for the Change Variable Type dataset shows that 85% of the importance is absorbed by the OO metrics (Figure 3). The CFG embedding features consistently ranked at the bottom of the importance hierarchy, confirming that the models mostly ignored the CFG embeddings when making final classification decisions. To investigate whether the predictive limi-

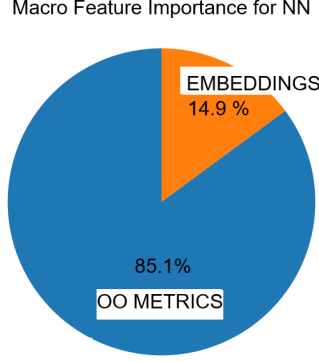


Fig. 3 Macro Feature Share for the Neural Network model on the *Change Variable Type* dataset, showing the overwhelming reliance on OO metrics over the embeddings.

tations of our embedding features were tied to the CFG representation, we expanded our framework to include GraphCode2Vec. It is important to note that this evaluation was scoped to a subset of the original software projects and five refactoring operations. This restriction was necessary because GraphCode2Vec processes compiled bytecode, which required compiling each project. Consequently, the ablated (OO metrics only) baselines were recomputed exclusively on this reduced corpus, which accounts for the slight numerical variations observed when compared to the ablated configurations reported in Table 7. The five selected refactoring techniques were chosen specifically because they contained a sufficient number of positive instances to ensure meaningful model training and prediction evaluation.

Unlike the Joern + LINE paradigm, where the XGBoost model trained on the ablated feature space systematically outperformed the full hybrid model, the introduction of GraphCode2Vec embeddings shows that combining traditional OO metrics with dense representation vectors can preserve or actively enhance predictive capabilities. As shown in the ablation study reported in Table 8 the hybrid feature space avoids the destructive noise injection observed with the CFG embeddings, allowing the XGBoost (Full) model to either match the ablated baseline or yield a performance gain, with the exception of the Move Method operation.

For refactoring targets such as Change Return Type and Add Parameter, the XGBoost Full model outperformed its ablated counterpart. The NN Full model showed analogue results for the Move Method operation. For Change Return Type, the full XGBoost model reaches an F1-score of 0.77 (compared to 0.74 of the Ablated space) for the positive class. This behavior suggests that these algorithms are not degraded by dense latent spaces.

However the integration of GraphCode2Vec features is not universally beneficial, and the data highlights instances of performance degradation. The most prominent regression is observed in the Move Method task, where the addition of embeddings causes the XGBoost F1-score to drop significantly from 0.83 in the Ablated space to 0.76 in the hybrid feature space. This contrasted to the results obtained by the NNs,

Table 8 GraphCode2Vec Evaluation: Predictive Performance (Positive Class / Refactored)

Refactoring Technique	XGBoost (Full)			Neural Network (Full)			XGBoost (Ablated)			NN (Ablated)			NN Only Embeddings		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Rename Method	0.70	0.73	0.71	0.71	0.68	0.70	0.69	0.73	0.71	0.62	0.78	0.69	0.63	0.53	0.58
Add Parameter	0.77	0.76	0.76	0.72	0.76	0.74	0.72	0.77	0.74	0.75	0.83	0.79	0.65	0.89	0.75
Change Return Type	0.78	0.76	0.77	0.71	0.68	0.70	0.77	0.71	0.74	0.78	0.72	0.75	0.71	0.67	0.69
Extract Method	0.80	0.75	0.78	0.68	0.72	0.70	0.75	0.81	0.78	0.73	0.73	0.73	0.72	0.70	0.71
Move Method	0.75	0.77	0.76	0.70	0.80	0.74	0.81	0.86	0.83	0.66	0.78	0.72	0.72	0.70	0.71

while targeting the same refactoring operation, showing that when working on the same target, and using the same embeddings, results can vary depending on the model utilized.

Despite the data showing that the added embeddings do not always provide a gain, their impact is highlighted by the NN Only Embeddings configuration. When traditional OO metrics are removed from the data, the Neural Network trained exclusively on GraphCode2Vec vectors achieves robust performance levels. For instance, in the Extract Method task, the embedding-only model yields an F1-score of 0.71, remaining competitive with the models possessing the entire suite of features. With the exclusion of the Rename Method operation, the embedding-only model always achieved a F1-score of 0.69 or higher. These dynamics indicate that while CFG embeddings acted as noise, GraphCode2Vec features encapsulate useful information, allowing supervised classifiers to forecast refactoring operations even in the absence of traditional software metrics.

We have discussed the prediction performances of the models when utilizing the GraphCode2Vec embeddings, but it is important to verify whether these newly generated vectors truly contribute in driving the model classifications.

The macro analysis for the Neural Network’s (Figure 4) Add Parameter dataset reveals that 60% of the model’s total decision making weight is absorbed by the embeddings. This shows that when the dependencies from the PDG and the syntactic context from the BiLSTM are combined, the models leverage this joint information to predict refactoring operations. These findings remain consistent across all other datasets under study.

By combining the empirical results from the ablation studies, and the interpretability analyses of both the XGBoost model and the NN, we can formally articulate the answer to our second Research Question.

Macro Feature Importance for NN

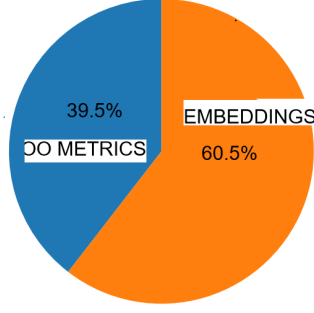


Fig. 4 Macro Feature Share for the Neural Network model on the *Add Parameter* dataset, showing the significant contribution of embeddings.

Answer to RQ2

The results reveal a contrast between the two alternative embedding representation strategies:

- **CFG Embeddings (Joern + LINE):** Suffered from feature noise and underutilization, leading to performance drop-offs for the XGBoost. The NN also failed to achieve an upward trend.
- **Multi-Modal Embeddings (GraphCode2Vec):** The use of Jimple IR and jointly embedded PDG and lexical token sequences enhanced performance in several tasks, and were able to drive the models predictions. The Neural Network trained exclusively on these vectors achieved competitive results.

4.3 Probability Calibration

While models can be evaluated on their predictive performance through the use of information retrieval metrics like Precision, Recall, and F1-score, it is also important to quantify the reliability of the probabilistic outputs. To evaluate the empirical reliability of the model’s predictions, it is necessary to transition toward scoring rules such as LogLoss and the Brier score [46].

As such, we report the Log Loss and Brier Score across each of the 5 core target refactoring datasets. This analysis focuses explicitly on the performance delta between the full hybrid feature space (with embeddings generated through GraphCode2Vec) and the ablated space relying solely on traditional OO metrics, as detailed in Table 9.

Probability calibration results show a task-dependent impact of embeddings rather than a uniform effect. In tasks like *Add Parameter*, embeddings improve reliability: XGBoost Full reduces the Brier Score from 0.1704 to 0.1564 ($\Delta = -8.22\%$) and NN Log Loss drops from 0.5409 to 0.4761 ($\Delta = -11.98\%$). Conversely, calibration degrades in *Move Method* and *Extract Method*. For *Move Method*, XGBoost Full experiences a

Table 9 Probability Calibration Comparison: Full Feature Space vs. Ablated OO Metrics

Dataset / Refactoring Task	XGBoost Model (LogLoss / Brier)			Neural Network Model (LogLoss / Brier)		
	Full	Ablated	$\Delta\%$	Full	Ablated	$\Delta\%$
Add Parameter						
LogLoss	0.4658	0.4985	-6.56%	0.4761	0.5409	-11.98%
Brier Score	0.1564	0.1704	-8.22%	0.1584	0.1639	-3.36%
Change Return Type						
LogLoss	0.4802	0.5020	-4.34%	0.6110	0.6497	-5.96%
Brier Score	0.1567	0.1649	-4.97%	0.2090	0.1869	+11.82%
Extract Method						
LogLoss	0.5209	0.4742	+9.85%	0.5996	0.6105	-1.79%
Brier Score	0.1626	0.1545	+5.24%	0.2015	0.2124	-5.13%
Move Method						
LogLoss	0.6145	0.4472	+37.41%	0.7641	0.8980	-14.91%
Brier Score	0.1831	0.1453	+26.02%	0.2199	0.2087	+5.37%
Rename Method						
LogLoss	0.6795	0.6339	+7.19%	0.6943	0.6988	-0.64%
Brier Score	0.2138	0.2105	+1.57%	0.1993	0.2146	-7.13%

severe breakdown (Log Loss: 0.4472 to 0.6145, $\Delta = +37.41\%$; Brier: 0.1453 to 0.1831, $\Delta = +26.02\%$) and NN Brier shifts to 0.2199 ($\Delta = +5.37\%$). For *Extract Method*, XGBoost Ablated outperforms the hybrid model (0.1545 vs. 0.1626, $\Delta = +5.24\%$).

To test for a generalized effect, a non-parametric two-tailed Wilcoxon signed-rank test was conducted on aggregated paired observations ($n = 10$ combining Log Loss and Brier Score) [48]. For XGBoost, the comparison yielded $W = 18.0$ ($p = 0.2754$), while the Neural Network resulted in $W = 12.0$ ($p = 0.0840$). Since both p -values exceed $\alpha = 0.05$, the null hypothesis cannot be rejected, confirming that embedding integration does not induce a statistically significant global shift in calibration.

4.4 Experimental Variants

To investigate whether the predictive limits of the Joern+LINE pipeline were artifacts of specific architectural choices or hyperparameter settings, we evaluated five targeted experimental variants. Complete tabular results across all positive and negative classes for these variants are archived in our replication package [15].

Stacking Ensemble (Late Fusion).

To leverage potential synergies between feature spaces, we trained a meta-learner using out-of-fold probability predictions from an OO-metric expert (XGBoost) and an embedding expert (Neural Network). Feature importance analysis of the meta-learner revealed a near-total reliance on the OO-metric expert, effectively discounting the Neural Network’s predictions. Consequently, the ensemble achieved F1-scores virtually identical to the ablated XGBoost baseline across all tasks (e.g., 0.74 vs. 0.74 on Rename Method; 0.82 vs. 0.84 on Change Variable Type), confirming that late-fusion stacking cannot compensate for noisy embedding inputs.

Dimensionality Reduction.

We hypothesized that the 64-dimensional latent space of LINE might introduce excessive noise. Applying Principal Component Analysis (PCA) to compress the embeddings into 3 principal components before XGBoost ingestion yielded no performance gains over the ablated baseline (e.g., Extract Variable positive F1 dropped from 0.78 to 0.73; Move Method remained flat at 0.80/0.81). This indicates that linear variance reduction does not isolate a cleaner predictive signal from CFG topologies.

Enriched Topological Representation (CPGs).

To test if Control Flow Graphs were representationally insufficient, we substituted CFGs with Code Property Graphs, which unify AST, CFG, and PDG structures via Joern. Despite this substantial structural enrichment, classification accuracy remained unchanged compared to the baseline CFG pipeline, demonstrating that graph density alone does not overcome the noise issue when raw source code topologies are evaluated without intermediate representations.

Latent Space Resolution ($d \in \{32, 128\}$).

Varying the LINE embedding vector dimensions to 32 and 128 produced no statistically significant deviation in F1-scores compared to the standard 64-dimensional configuration. Across evaluated tasks, XGBoost Full performance remained bounded between 0.66 and 0.75 F1, indicating that topological underperformance is intrinsic to the source-level CFG graph representation rather than a vector capacity bottleneck.

Self-Supervised Learning Paradigms.

Finally, we explored Autoencoding and Contrastive Learning pre-training to address potential data scarcity issues in deep architectures:

- **Autoencoding Initialization:** Reconstruction loss converged during pre-training, but fine-tuning on the classification task yielded F1-scores predominantly inferior to standard supervised Neural Networks (e.g., *Extract Variable* positive F1 dropped severely to 0.32).
- **Contrastive Learning Initialization:** Contrastive pre-training using data augmentations consistently suffered from representation collapse, rendering it incompatible with our tabular-embedding feature space.

5 Discussion and Lessons Learned

Our empirical evaluation frames code smell refactoring prediction as a supervised binary classification task across diverse machine learning models and representation strategies. The findings offer three primary lessons:

- **The predictability of a refactoring depends on its driving factors.** The empirical findings establish a distinction between structural transformations and semantic changes. Operations restricted to method signatures or local statements (e.g., Add Parameter, Change Variable Type) are easier to predict when they strongly correlate with measurable code complexity and size metrics. Conversely,

operations that represent conceptual changes (e.g., Rename Method) are not triggered by quantifiable code complexity, but are instead driven by human intent or domain-driven design decisions. Because static and topological features can only capture a partial signal of these subjective choices, purely quantitative models face a predictability barrier when isolated from the developer’s development context. While these results are not definitive, they point to a limitation in features extracted solely from the codebase when predicting operations like method renames. Effectively predicting these tasks likely requires contextual meta-information outside the source code layer to model the underlying domain logic.

- **Source code derived CFGs introduce feature noise without improving performance.** When combined with CFG embeddings, OO metrics are the main driver of models’ predictions, and the 64-dimensional embedding space generated via LINE provides no measurable predictive gain. This lack of performance can be hypothesized to stem from the fact that a CFG captures only the execution order of statements, making the graph topology sensitive to variations like code formatting or statement reordering, caused by different programming styles, that alter graph topologies without changing underlying semantics [49]. In the case of tree-based models, this introduced noise could make it harder for the algorithm to select effective splits and isolate useful patterns.
- **The combination of Jimple IR and richer graph modeling improves refactoring predictions.** Transitioning from CFGs to a combination of PDG and sequential token syntax representations shows a performance increase in both XGBoost and NN. However, because this transition changes both the abstraction layer (Jimple IR) and the architecture (PDG, Word2Vec, and BiLSTM), their individual contributions cannot be isolated under the methodology of this study.

Takeaways for Practitioners.

Predictive refactoring tools should not be deployed as one-size-fits-all solutions. Concrete automation should be possible for localized, metric-driven refactorings (Add Parameter, Change Return Type), whereas conceptual tasks (Rename Method) require human-in-the-loop oversight. Leveraging these predictive tools is not about replacing human critical thinking, but about optimizing developers’ limited time and resources.

Takeaways for Researchers.

This study showed the necessity of adequate syntactic and topological modeling to accurately anticipate software refactoring needs. Increasing topological complexity on raw source code (e.g., rich CPGs via Joern) did not translate into informational gains in our study; while moving to an intermediate representation (Jimple IR) that jointly integrates control and data dependencies with lexical token sequences, enhanced predictive accuracy compared to baseline CFGs. Future research should focus on exploring different topological paradigms and how these interact with varied intermediate representations of code, to better isolate program logic.

6 Threats to Validity

The study presented in this paper is subject to several limitations and threats to its validity. To describe the threats to validity, we followed the structure proposed by Runeson et al. [50], organizing them in terms of construct validity, external validity, and reliability. Since we did not investigate causal relationships, internal validity is not discussed [50].

Construct Validity.

- **Ground Truth Accuracy and Label Noise:** Target labels rely on automated extraction via RefactoringMiner [9]. While extraction errors could introduce label noise, the tool is the established de facto standard in empirical software engineering [8, 21, 22], exhibiting an evaluated precision of 99.6% and recall of 94%.
- **Static Tool Calibration:** Static analysis tools can exhibit divergent metric thresholds [51]. We mitigated this by combining two independent analysis tools (Designite [10] and the CK Analysis Tool), ensuring feature extraction was not bound to a single rule-based engine.
- **Refactoring Intent & Quality:** Developers’ refactoring actions do not guarantee that targeted code smells were effectively resolved. We mitigated this by analyzing snapshots across a two-year delta, deterministically retaining only instances where Designite confirmed the smell was eliminated. This validation step was applied on an aggregated dataset of over 8000 refactoring operations, spanning 79 different projects. Statistically, evaluating dozens of diverse commit histories smooths out isolated sub-optimal choices, ensuring that the models capture positive actions.

Reliability.

- **Replicability & Commit Versioning:** To mitigate potential variations in repository commit lifespans, we provide a complete, open-access replication package containing all processed tabular datasets, embedding vectors, and execution pipelines [15].

External Validity.

- **Language Specificity:** The pipeline was calibrated for Java, meaning findings regarding structural metric dominance and CFG noise cannot be directly generalized to dynamically typed languages (e.g., Python, JavaScript).
- **Open-Source vs. Industrial Codebases:** All analyzed projects were open-source repositories from GitHub. To ensure project maturity and industrial relevance, we utilized the validated dataset by Aniche et al. [8]. The authors described their sample as containing numerous industrial-scale projects spanning diverse domains, a property directly inherited by our study. We reinforced this mitigation by exclusively sampling the 79 projects exhibiting the highest volume and density of historical refactoring activities. This filtering process guaranteed the exclusion of small-scale repositories, isolating ecosystems subject to continuous maintenance cycles, that approximate as closely as possible industrial environments.

- **Dataset Balancing:** The test set was balanced (1:1 ratio) via undersampling to evaluate model capacity without majority-class bias [8]. Reported metrics reflect baseline discriminative capability rather than production precision under naturally skewed refactoring distributions.

7 Related Work

The rapid advancements in machine learning and the integration of data driven approaches into software maintenance have shifted the focus of research from static, rule based analysis tools, toward recommendation systems. As such a substantial body of ongoing work now seeks to automate code maintenance.

7.1 Empirical Studies on Code Smells Refactoring

To build effective automated recommendation systems, it is important to understand how developers operate in production environments. An extensive body of empirical literature has investigated whether the presence of code smells triggers corrective transformations, revealing a divergence between static analysis recommendations and practitioners’ behavior.

To examine the difference between theory and practice, Yoshida et al. [52] analyzed the emergence of code smells relative to the actual refactoring patterns employed by developers and they showed that developers rarely apply the theoretically appropriate refactoring patterns to mitigate smells, with only 26.8% of class level and 16.0% of method level anomalies being resolved using the textbook remedy.

When developers do attempt manual remediation, they often bundle multiple operations into compound transformations, which amplifies the risk of human error. Bibiano et al. [21] analyzed 4,607 instances of such batch refactorings across 57 Java projects using RefactoringMiner, discovering that 51% of the analyzed batches ended up introducing entirely new code smells, while 38% failed to remove existing ones.

Refactoring operational complexity is further compounded when observing code evolution over long lifecycles. Shetty and Sharma [22] executed a longitudinal study tracking 212,664 commits from 87 open-source Java projects via Designite and RefactoringMiner. Their work revealed prevalence of dangling instances of smells: approximately 82% of code smells are removed incidentally through general feature changes or bug fixes rather than explicit refactorings, and over half of all intentional refactorings do not actually resolve any underlying smells.

These findings highlight a limitation in traditional static analysis, the presence of a code smell does not always lead to the execution of a refactoring intervention, and when it does and a manual remediation is attempted, the lack of tool guidance can lead to the introduction of new defects.

7.2 ML Embedding Application on Software Quality Assessment

Traditional software quality assessment usually relies on evaluating different metrics. The use of these metrics fails to capture text semantics embedded within source code,

as well as the sequential nature of program instructions and their data and control flow.

For instance, regarding textual artifacts, Flisar and Podgorelec [53] explored the utility of Natural Language Processing in software maintenance by classifying Self-Admitted Technical Debt (SATD) within source code comments, combining Information Gain metrics with dense Word2Vec embeddings. Moving beyond isolated comment analysis, Huo et al. [54] proposed a different approach through their Comments Augmented Programs-Convolutional Neural Network (CAP-CNN) architecture for software defect prediction. This framework utilizes dual CNN to jointly embed token sequences from source code, statement structures, and natural language comments. Both studies showed the importance of capturing textual and syntactic semantics, and how these can be combined with other metrics to enhance models' predictability.

Beyond textual representations, embeddings can be used to capture the sequential execution, and control flow of a program. To achieve this, Huo et al. [55] introduced Control Graph-CNN (CG-CNN), a deep learning framework designed for automated bug localization that shifts the analytical focus onto execution paths of code.

The use of embeddings has also been extended to code smell detection. Nizam et al. [56] developed a generic embedding optimization framework leveraging contrastive learning to refine existing pre-trained models. Their results produced a specialized CodeBERT model optimized for code smell detection.

Finally, we cite our previous work on the use of large language models for the detection and explainability of architectural smells [57], a symptom of a different type of TD known as Architectural Technical Debt, which has been examined in prior studies [58–60].

Our solution shares the core objective of leveraging embedding spaces to surpass the limitations of traditional metrics. While the aforementioned studies focus either on sequential token text or modeling execution paths via CFGs, our framework integrates traditional OO metrics with both sequential syntax of the code and PDG topologies, to produce an accurate representation of the program logic.

7.3 Approaches to Automated Software Refactoring Recommendations

One significant study in this area of research was conducted by Aniche et al. [8]. They evaluated the effectiveness of several supervised machine learning algorithms to recommend 20 distinct types of refactoring operations at the class, method, and variable levels. Their approach is guided by traditional OO metrics. They formulated the problem as an independent binary classification task to determine whether a specific element needs a particular refactoring or not, and relies on traditional models, such as Random Forest, trained on over two million instances.

This study served as the direct inspiration and baseline for our research. In contrast to their purely metric-driven setup, our framework extends the predictive pipeline into dense embedding spaces. Our methodology integrates these traditional OO metrics with embeddings. This allows us to evaluate whether a hybrid feature space can enhance models' predictive capabilities.

Other approaches that incorporate embeddings for richer representations include the work from Palit et al. [61]. The authors proposed a deep learning pipeline to identify Extract Method candidates. Instead of using OO metrics, they leveraged a pre-trained GraphCodeBERT model to capture the syntactic and semantic properties of the source code. The embeddings were then compressed using an Autoencoder and served as input to a Random Forest binary classifier.

Similarly, Zhang et al. [62] introduced GMove, a tool designed to recommend Move Method refactorings by combining the semantic and structural features with traditional metrics. Their tool utilizes Code2Vec to extract structural representations from ASTs and employs a Convolutional Neural Network architecture to perform binary classification for Feature Envy detection.

Recent work has also explored prompt-based generative paradigms. Pandini et al. [63] evaluated Large Language Models to support the automated refactoring of a type of architectural smell, Cyclic Dependency, across Java, C#, and Python.

Finally, a search-based framework for automated refactoring was proposed by Liu et al. [64]. Their framework was designed to minimize inconsistencies between implemented code and its expected architectural design. It formulates refactoring as a multi-objective optimization task, applying a strict set of 32 Java-specific pre-conditions to guarantee code executability.

Although all these approaches have shown promising results in automated refactoring recommendations, they all suffer from distinct limitations. Prompt-based solutions, like the one from Pandini et al. [63], depend on third party Large Language Models, and are not easily scalable. The ML frameworks shown are either limited to traditional OO metrics, like the case of Aniche et al. [8] work, or are restricted in scope to a single, localized refactoring operation; this is the case for Palit et al. [61] and Zhang et al. [62] solution. Our framework introduces an architecture trained to predict 10 different refactoring operations. It utilizes a hybrid representation comprising both OO metrics and embedding space.

8 Conclusions and Future Work

This study investigated the viability of the use of ML models in predicting refactoring opportunities on code affected by code smell. By framing code smell refactoring prediction as a supervised binary classification task, we benchmark a diverse set of ML models and architectures, while moving beyond the use of just traditional OO metrics toward hybrid feature space, with the addition of vector embeddings. Based on our empirical findings, the core research questions are addressed as follows

- **Answer to RQ1:** Supervised ML algorithms are reliable predictors of refactoring tasks, with XGBoost and NN models consistently achieving higher predictive accuracy over baselines. However, the driving factors of each refactoring technique impact their predictability.
- **Answer to RQ2:** Traditional OO metrics act as the primary driving force, when paired with CFG embeddings, as the latter introduces feature noise. Transitioning

to the GraphCode2Vec framework inverts this dynamic: the multi-modal embeddings account for the majority of the overall feature importance, while providing a complementary and independent predictive signal.

These answers led to two main conclusions. First, a difference in predictability exists between operations driven mainly by human intent or domain design choices and transformations that correlate with structural complexity, with the latter showing better predictability. This suggests a potential limitation in using features extracted solely from the codebase. Second, the selection of the code abstraction layer and the specific modeling of structural and lexical representations has an impact on the models performance.

In summary, this research shows that predicting refactorings through ML models is viable, though the predictive accuracy varies depending on how the source code is processed and transformed, and the nature of each targeted refactoring operation.

Several directions remain open for future research, including evaluating the pipeline under naturally skewed class distributions and extending it to additional languages such as Python and C++; conducting controlled experiments to determine whether the gains from GraphCode2Vec over Joern + LINE stem from the abstraction layer (Jimple bytecode vs. raw source) or the richer structural architecture (PDGs and BiLSTMs vs. CFGs); enriching the graph representation with node-level software metrics extracted via static analysis, shifting toward an attributed graph abstraction for potentially higher predictive accuracy; and incorporating repository process metrics (e.g., code churn, developer experience, commit and PR discussions) alongside business-level metadata, such as architectural requirements and domain-driven design boundaries, to capture the context needed to predict semantic refactorings like method renames.

References

- [1] Avgeriou, P., Ozkaya, I., Chatzigeorgiou, A., Ciolkowski, M., Ernst, N.A., Koontz, R.J., Poort, E., Shull, F.: Technical debt management: The road ahead for successful software delivery. In: 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE), pp. 15–30 (2023). IEEE
- [2] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code vol. 1. Addison-Wesley, Reading, MA (1999)
- [3] Lanza, M., Marinescu, R.: Object-oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems. Springer, Berlin, Heidelberg (2006)
- [4] Arcelli Fontana, F., Mäntylä, M.V., Zaroni, M., Marino, A.: Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* **21**(3), 1143–1191 (2016)
- [5] AlOmar, E.A., Mkaouer, M.W., Ouni, A.: Behind the intent of extract method refactoring: A systematic literature review. *IEEE Transactions on Software*

- [6] Bavota, G., De Lucia, A., Marcus, A., Oliveto, R.: Recommending refactoring operations in large software systems. In: *Recommendation Systems in Software Engineering*, pp. 387–419. Springer, Berlin, Heidelberg (2013)
- [7] Cui, D., Wang, S., Luo, Y., Li, X., Dai, J., Wang, L., Li, Q.: Rmove: Recommending move method refactoring opportunities using structural and semantic representations of code. In: *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 281–292 (2022). IEEE
- [8] Aniche, M., Maziero, E., Durelli, R., Durelli, V.H.: The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering* **48**(4), 1432–1450 (2020)
- [9] Tsantalis, N., Ketkar, A., Dig, D.: Refactoringminer 2.0. *IEEE Transactions on Software Engineering* **48**(3), 930–950 (2020)
- [10] Sharma, T.: Designite: A software design quality assessment tool. In: *Proceedings of the 38th International Conference on Software Engineering Companion*, pp. 681–684 (2016). ACM
- [11] Aniche, M.: CK Tool. GitHub (2019). <https://github.com/mauricioaniche/ck>
- [12] Yamaguchi, F., Lottmann, N., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: *2014 IEEE Symposium on Security and Privacy (S&P)*, pp. 590–604 (2014). IEEE
- [13] Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., Mei, Q.: Line: Large-scale information network embedding. In: *Proceedings of the 24th International Conference on World Wide Web* (2015)
- [14] Ma, W., Zhao, M., Soremekun, E., Hu, Q., Zhang, J.M., Papadakis, M., Cordy, M., Xie, X., Traon, Y.L.: Graphcode2vec: Generic code embedding via lexical and program dependence analyses. In: *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 524–536 (2022)
- [15] Lanza, A., Bochicchio, M., Arcelli Fontana, F.: Dataset and Replication Package. Zenodo (2026). <https://doi.org/10.5281/zenodo.22814293> . <https://doi.org/10.5281/zenodo.22814293>
- [16] Sharma, T., Singh, P., Spinellis, D.: An empirical investigation on the relationship between design and architecture smells. *Empirical Software Engineering* **25**(5), 4020–4068 (2020)
- [17] Uchôa, A., Barbosa, C., Oizumi, W., Blenfilio, P., Lima, R., Garcia, A., Bezerra, C.: How does modern code review impact software design degradation? an

- in-depth empirical study. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 511–522 (2020). IEEE
- [18] Alenezi, M., Zarour, M.: An empirical study of bad smells during software evolution using designite tool. *i-Manager’s Journal on Software Engineering* **12**(4), 12 (2018)
 - [19] Eposhi, A., Oizumi, W., Garcia, A., Sousa, L., Oliveira, R., Oliveira, A.: Removal of design problems through refactorings: Are we looking at the right symptoms? In: 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), pp. 148–153 (2019). IEEE
 - [20] Sharma, T., Fragkoulis, M., Spinellis, D.: House of cards: Code smells in open-source c# repositories. In: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 424–429 (2017). IEEE
 - [21] Bibiano, A.C., Fernandes, E., Oliveira, D., Garcia, A., Kalinowski, M., Fonseca, B., Oliveira, R., Oliveira, A., Cedrim, D.: A quantitative study on characteristics and effect of batch refactoring on code smells. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–11 (2019). IEEE
 - [22] Shetty, G., Sharma, T.: Mapping code smells and refactorings accurately: Insights from an empirical study. In: 2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–11 (2025). IEEE
 - [23] Yaraghi, A.S., Holden, D., Kahani, N., Briand, L.: Automated test case repair using language models. *IEEE Transactions on Software Engineering* (2025)
 - [24] Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* **20**(6), 476–493 (1994)
 - [25] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, 2nd edn. Pearson Education, Boston, MA (2006)
 - [26] Horwitz, S., Reps, T.: The use of program dependence graphs in software engineering. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **14**(3), 398–433 (1992)
 - [27] Bengio, Y., Courville, A., Vincent, P.: Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence* **35**(8), 1798–1828 (2013)
 - [28] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: *Proceedings of the 1999 CASCON*

- Conference, p. 13 (1999)
- [29] Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space. In: Proceedings of the International Conference on Learning Representations (ICLR) (2013)
 - [30] Graves, A., Schmidhuber, J.: Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks* **18**(5-6), 602–610 (2005)
 - [31] Tantithamthavorn, C.K., Jiarapakdee, J.: Explainable ai for software engineering. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1–2 (2021). IEEE
 - [32] Silva, D., Tsantalis, N., Valente, M.T.: Why we refactor? confessions of github contributors. In: Proceedings of the 2016 24th Acm Sigsoft International Symposium on Foundations of Software Engineering, pp. 858–870 (2016)
 - [33] Pantiuchina, J., Zampetti, F., Scalabrino, S., Piantadosi, V., Oliveto, R., Bavota, G., Di Penta, M.: Why developers refactor source code: A mining-based study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **29**(4) (2020)
 - [34] Micci-Barreca, D.: A preprocessing scheme for high-cardinality categorical attributes in classification and prediction problems. *ACM SIGKDD Explorations Newsletter* **3**(1), 27–32 (2001)
 - [35] Bishop, C.M., Nasrabadi, N.M.: *Pattern Recognition and Machine Learning* vol. 4. Springer, Berlin, Heidelberg (2006)
 - [36] Fenton, N., Bieman, J.: *Software Metrics: A Rigorous and Practical Approach*, 3rd edn. CRC Press, Boca Raton, FL (2014)
 - [37] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., *et al.*: Scikit-learn: Machine learning in python. *Journal of machine learning research* **12**(Oct), 2825–2830 (2011)
 - [38] Cortes, C., Vapnik, V.: Support-vector networks. *Machine Learning* **20**(3), 273–297 (1995)
 - [39] Chen, T., Guestrin, C.: Xgboost: A scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 785–794 (2016). ACM
 - [40] Goodfellow, I., Bengio, Y., Courville, A.: *Deep Learning*. MIT press (2016)
 - [41] Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization.

- [42] Kohavi, R.: A study of cross-validation and bootstrap for accuracy estimation and model selection. In: International Joint Conference on Artificial Intelligence (IJCAI), vol. 14, pp. 1137–1145 (1995)
- [43] Snoek, J., Larochelle, H., Adams, R.P.: Practical bayesian optimization of machine learning algorithms. In: Advances in Neural Information Processing Systems (NeurIPS), vol. 25 (2012)
- [44] Prechelt, L.: Early stopping-but when? In: Neural Networks: Tricks of the Trade, pp. 55–69. Springer, Berlin, Heidelberg (2002)
- [45] Lundberg, S.M., Lee, S.-I.: A unified approach to interpreting model predictions. In: Advances in Neural Information Processing Systems vol. 30. Curran Associates, Inc., Red Hook, NY (2017)
- [46] Zadrozny, B., Elkan, C.: Obtaining calibrated probability estimates from decision trees and naive bayesian classifiers. In: Icml, vol. 1, p. 2001 (2001)
- [47] Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., Dennison, D.: Hidden technical debt in machine learning systems. *Advances in neural information processing systems* **28** (2015)
- [48] Wilcoxon, F.: Individual comparisons by ranking methods. In: Breakthroughs in Statistics: Methodology and Distribution, pp. 196–202. Springer, Berlin, Heidelberg (1992)
- [49] Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C.: A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* **51**(4), 1–37 (2018)
- [50] Runeson, P., Höst, M., Rainer, A., Regnell, B.: Case Study Research in Software Engineering: Guidelines and Examples. John Wiley & Sons (2012)
- [51] Lefever, J., Cai, Y., Cervantes, H., Kazman, R., Fang, H.: On the lack of consensus among technical debt detection tools. In: Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 121–130. IEEE, Piscataway, NJ (2021)
- [52] Yoshida, N., Saika, T., Choi, E., Ouni, A., Inoue, K.: Revisiting the relationship between code smells and refactoring. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC). IEEE, Piscataway, NJ (2016)
- [53] Flisar, J., Podgorelec, V.: Enhanced feature selection using word embeddings for self-admitted technical debt identification. In: 2018 44th Euromicro Conference on

Software Engineering and Advanced Applications (SEAA), pp. 230–233 (2018). IEEE

- [54] Huo, X., Yang, Y., Li, M., Zhan, D.-C.: Learning semantic features for software defect prediction by code comments embedding. In: 2018 IEEE International Conference on Data Mining (ICDM), pp. 1049–1054. IEEE, Piscataway, NJ (2018)
- [55] Huo, X., Li, M., Zhou, Z.-H.: Control flow graph embedding based on multi-instance decomposition for bug localization. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 34, pp. 4223–4230 (2020)
- [56] Nizam, A., İslamoğlu, E., Adali, Ö.K., Aydin, M.: Optimizing pre-trained code embeddings with triplet loss for code smell detection. *IEEE Access* **13**, 31335–31350 (2025)
- [57] Tessa, C., Bochicchio, M., Arcelli Fontana, F.: Exploring architectural smells detection through llms. In: European Conference on Software Architecture, pp. 90–98 (2025). Springer
- [58] Bochicchio, M., Sas, D., Gilardi, A., Arcelli Fontana, F.: An empirical study on architectural smells through a pipeline for continuous technical debt assessment. *Information and Software Technology* **185**, 107783 (2025)
- [59] Bochicchio, M., Walter, B., Arcelli Fontana, F.: Empirical study on the correlation among architectural smells. *Information and Software Technology* **199**, 108283 (2026)
- [60] Messa, A., Bochicchio, M., Arcelli Fontana, F.: Exploring the impact of architectural smells refactoring in microservice projects. In: 2026 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 1–6 (2026). IEEE
- [61] Palit, I., Shetty, G., Arif, H., Sharma, T.: Automatic refactoring candidate identification leveraging effective code representation. In: 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 369–374 (2023). IEEE
- [62] Zhang, Y., Gu, Z., Zhang, N., Zheng, K.: Recommending move method refactoring opportunities based on feature fusion and deep learning. *IEEE Latin America Transactions* **24**(2), 135–143 (2026)
- [63] Pandini, G., Martini, A., Videsjorden, A.N., Arcelli Fontana, F.: An exploratory study on architectural smell refactoring using large languages models. In: 2025 IEEE 22nd International Conference on Software Architecture Companion (ICSAC), pp. 462–471 (2025). IEEE

- [64] Liu, J., Jin, W., Zhou, J., Feng, Q., Fan, M., Wang, H., Liu, T.: 3erefactor: Effective, efficient and executable refactoring recommendation for software architectural consistency. *IEEE Transactions on Software Engineering* (2024)