# ROSE: Transformer-Based Refactoring Recommendation for Architectural Smells

Samal Nursapa
*Mälardalen University*
Västerås, Sweden

Anastassiya Samuilova
*Mälardalen University*
Västerås, Sweden

Alessio Bucaioni
*Mälardalen University*
Västerås, Sweden
alessio.bucaioni@mdu.se

Phuong T. Nguyen
*University of L'Aquila*
L'Aquila, Italy
phuong.nguyen@univaq.it

***Abstract*—Architectural smells, design flaws such as God Class, Cyclic Dependency, and Hub-like Dependency, erode maintainability and often impair runtime behaviour. While existing detectors flag these issues, they rarely suggest how to remove them. We developed ROSE, a recommender system that turns smell reports into concrete refactoring advice by leveraging pre-trained code transformers. We frame remediation as a three-way classification task (Extract Method, Move Class, Pull Up Method) and fine-tune CodeBERT and CodeT5 on 2.1 million refactoring instances mined with RefactoringMiner from 11,149 open-source Java projects. Running with ten-fold cross-validation, CodeT5 gets 96.9% accuracy and a macro-F1 of 0.95, outperforming CodeBERT by 10 percentage points and all classical baselines reported in the original dataset study. Confusion-matrix analysis shows that both models separate Pull Up Method well, whereas Extract Method remains challenging because of overlap with structurally similar changes. These findings provide the first empirical evidence that transformers can close the gap between architectural-smell detection and actionable repair. The study illustrates the promise, and current limits, of data-driven, architecture-level refactoring, laying the groundwork for richer recommender systems that cover a wider range of smells and languages. We release code, trained checkpoints, and the balanced dataset under an open licence to encourage replication.**

***Index Terms*—Architectural Smells, Software Refactoring, Transformer Models, CodeBERT, CodeT5, Refactoring Recommendation.**

## I. Introduction

Software architecture is pivotal to the maintainability, scalability, and performance of modern applications [27]. Poor architectural decisions frequently give rise to architectural smells—design flaws that manifest above the code-element level, disturbing modularity, dependencies, and runtime behaviour [21]. Representative smells include God Class, Cyclic Dependency, and Hub-like Dependency [7]. Static-analysis tools such as Designite [28] and Arcan [6] detect these smells effectively, yet offer little guidance on how to remove them. Practitioners must still determine suitable refactorings manually, relying on qualitative reasoning or hand-crafted rules that struggle with the non-linear relationships between design flaws and quality attributes [21]. Recent deep-learning studies show promise for code-quality tasks: pre-trained Transformers outperform metric-based methods in code-smell detection [3] and can localise refactor-prone code spans [15]. However, to the best of our knowledge, no prior work has leveraged Transformer models to recommend concrete refactorings for architectural smells—a gap this study addresses.

Existing work on refactoring prediction employed classic classifiers over large code-metric datasets, forecasting whether a refactoring would occur [4]. More recent approaches use commit-message text or LSTMs to suggest fixes for a few code smells [23], yet none of them combines architectural-smell detection with pre-trained code Transformers to prescribe *which* refactoring to apply. Meanwhile, large language models (LLMs) can autonomously refactor simple code, but still falter on complex design issues such as dependency cycles [9]. These trends indicate that Transformer-based refactoring is emerging, whereas architectural smells remain under-served, triggering a timely research opportunity for empirical software engineering. To the best of our knowledge, *so far no tool has been developed for the recommendation of refactoring operations from architectural smells.*

In this work, we aim to bridge such a gap by developing ROSE, a Transformer-based framework to recommend **R**efactoring **O**perations from architectural **S**m**E**lls. Formulated as a multi-class classification problem, the approach fine-tunes two pre-trained models, i.e., CodeBERT and CodeT5 using both source-level representations and architectural-dependency features. An empirical evaluation was conducted on ROSE using a curated subset of more than 2M historical Refactoring Operations spanning 11,149 Java projects. Specifically, our goal is to answer the following Research Questions (RQs):

- **RQ**$_1$: *Are the considered Transformer models capable of correctly predicting the refactoring type required to repair a given architectural smell?* With this research question, we investigate whether the considered transformer method is able to recommend a correct refactoring starting from an architectural smell.
- **RQ**$_2$: *How do* CodeBERT *and* CodeT5 *differ in effectiveness and computational cost for this task?* We compare the two models to study which of them provides a better performance with respect both to the accuracy and computing resource.
- **RQ**$_3$: *Which architectural smells benefit most from learning-based refactoring recommendations?* This RQ ascertains which smell can be refactored most effectively with transformers. This is important in practice, as it helps developers choose a suitable recommendation technique

for a specific smell.

Our work makes the following key contributions:

- **ARCH-T5/BERT framework**. We develop the first Transformer pipeline to recommend refactorings for architectural smells.
- **Large-scale evaluation**. An empirical comparison of CODEBERT, CODET5, metric-based, and classical ML baselines has been conducted on a set of more than 3M balanced instances.
- **Emerging evidence**. Initial results show $>10$ pp F-score improvement over state-of-the-art baselines, demonstrating the feasibility of learning architectural-refactoring knowledge from big-code.
- **Open Science**. An anonymized replication package including source code, data splits, and trained model checkpoints, is openly archived on GitHub to facilitate future research at https://anonymous.4open.science/r/archsmell_transformers-66F8.

The remainder of this work is organized as follows. Section II reviews the related work. The proposed approach is presented in Section III. Afterward, Section IV reports and analyzes the experimental results. In Section V, we discuss the findings, and highlight the threats to validity. Finally, Section VI sketches future work, and concludes the paper.

## II. RELATED WORK

Existing research has made substantial progress in detecting code and architectural smells, optimising refactoring plans through search-based techniques, and applying machine learning to predict where refactoring is likely to occur. Nevertheless, a key gap persists: prior work rarely delivers end-to-end, data-driven recommendations that specify which refactoring operation should resolve a detected architectural smell. No study combines large-scale empirical evidence with modern Transformer models to close this decision loop. In contrast, our approach frames architectural-smell remediation as a multi-class prediction task and fine-tunes CodeBERT and CodeT5 on more than two million real refactoring instances from 11 149 projects, transforming code understanding into actionable, context-aware architectural guidance.

**Code-Smell Detection and Analysis.** Early investigations established basic terminology and catalogued detection methods. Zhang et al. [35] synthesised 39 primary studies, laying a conceptual foundation. Aldallal [1] extended the review to 47 studies and observed that most relied on small, single-project datasets and metric thresholds, limiting generalisability and actionable insight. Later surveys widened the scope. Singh et al. [29] mapped both smells and anti-patterns, while Santos et al. [26] introduced the smell effect, showing how smells influence downstream activities and pointing out evaluator subjectivity. Fernandes et al. [12] and Rasool and Arif [25] compared industrial tools such as SonarQube, JDeodorant and Designite, classifying them by threshold, rule, and graph strategies. They found little use of machine learning, limiting adaptability across languages and frameworks. Data-driven detectors now dominate. Classical machine-learning models like Random Forest and SVM lift accuracy on labelled datasets [8]. Deep networks (CNN, RNN, GNN) further reduce reliance on handcrafted metrics [22]. Transformer-based systems raise performance again: SCSmell stacks BERT variants [34]; RABERT adds relational bias for God-Class detection [3]; RefactorBERT identifies refactor-prone regions [15]. These approaches, however, focus on detection and seldom address architectural smells such as cyclic dependencies. Our study moves from detection to remediation. We fine-tune pre-trained Transformers on more than two million historical refactorings to predict the specific operation that resolves each architectural smell, closing a gap highlighted across prior surveys.

**Search-Based Refactoring.** Search-based software engineering frames refactoring as optimisation. Sequences of Fowler-style operations [13] evolve under genetic and swarm heuristics to improve cohesion, coupling and other metrics. Mariani et al. [19] identified genetic algorithms as dominant; Mohan and Greer [20] catalogued tools and noted growth in multi-objective search. Di Pompeo et al. [24] pushed the idea to architecture level with many-objective optimisation. These methods rely on fitness functions that are costly to tune, scale poorly with system size and ignore real developer practice. Our data-driven alternative learns directly from millions of recorded refactorings across eleven thousand projects, bypassing manual fitness engineering and providing context-aware guidance in constant time.

**Deep Learning for Smell Detection and Refactoring.** Deep networks supplement metric methods. Naik et al. [22] surveyed 17 studies using CNN, RNN, GNN and MLP for method-level prediction; gains were modest and language-specific. Alazba et al. [2] reviewed 67 studies on smell detection, dominated by clones and long methods. Malhotra et al. [18] showed hybrid RNN–CNN models improve precision but do not recommend repairs. Zhang et al. [36] reported dataset imbalance and inconsistent definitions. Transformers change the picture. SCSmell removes metric features, RABERT adds relational encoding, and RefactorBERT flags refactor-prone code. General-purpose language models can fix simple issues but struggle with architectural flaws [9]. None recommend a concrete refactoring for a given architectural smell. We frame remediation as multi-class prediction of the operation—Extract Class, Move Class, and others—using Transformers trained on real refactorings.

**Pre-trained Code Models.** Code-centric pre-trained models underpin many tasks. CodeBERT combines natural and programming language pairs [11]; GraphCodeBERT adds data-flow graphs [14]. Encoder–decoder families such as CodeT5 and CodeT5+ address generation and repair [31], while GPT-style models CodeGPT [17] and CodeRL [16] produce autoregressive code. No existing model predicts refactoring actions for architectural flaws. We repurpose CodeBERT and CodeT5 to fill this gap, demonstrating that their learned representations

enable data-driven architectural guidance.

**Architectural versus Code Smells.** Code smells receive extensive attention; architectural smells do not. De Paulo Sobrinho et al. [21] highlighted this imbalance. Fontana et al. [5] showed that removing cyclic dependencies and hubs improved response time by 47 percent and reduced memory by 20 percent, demonstrating runtime impact. Most work detects architectural smells or measures quality after refactoring but seldom links detection to repair. We combine identification with Transformer-based prediction of the most suitable refactoring, providing proactive architecture improvement.

**Transformer Models in Software Engineering.** Transformers underpin numerous code-intelligence tasks. CodeBERT excels at retrieval and summarisation; GraphCodeBERT improves clone detection; CodeT5 handles generation and repair. Xiao et al. [32] analysed 519 Transformer papers and noted challenges with compute cost and overfitting. Few studies aim at architectural design improvement or refactoring recommendation. We fine-tune CodeBERT and CodeT5 on more than two million refactorings, translating code understanding into actionable architectural advice.

**Datasets and Tool Support.** Zakeri-Nasrabadi et al. [33] found that fewer than half of the 45 smell datasets are public, with low project diversity. Detectors include SonarQube, Designite, Arcan, Sonargraph and Structure101. RefactoringMiner [10] mines fine-grained changes; PyRef serves Python. We release a corpus of 11,149 Java projects and more than two million refactorings, enriched with structural, process and ownership metrics, enabling Transformers to learn context and recommend repairs for architectural smells.

## III. Research methodology

To address the research questions, we conducted an empirical study that fine-tunes two transformer models, CodeBERT and CodeT5, to recommend refactorings for architectural smells [30]. These models are pre-trained with self-supervised objectives such as masked language modelling and therefore capture rich semantic and syntactic information from source code [11].

### A. Models

*a) CodeBERT:* CodeBERT is a transformer encoder trained on paired natural-language and source-code data across several programming languages. For this work, the model is fine-tuned as a multi-class classifier that maps a code fragment containing an architectural smell to one refactoring label drawn from a predefined set (for example Extract Method, Move Method, Pull Up Method). During fine-tuning the representation of the special classification token is passed through a feed-forward layer followed by softmax to obtain class probabilities.

*b) CodeT5:* CodeT5 adapts the encoder–decoder T5 architecture to programming languages and uses token- and span-masking during pre-training. Although originally designed for generation tasks, the encoder output can be used for classification. We therefore attach a linear classification head to the encoder, enabling prediction of the refactoring type required to resolve the detected smell.

### B. Dataset

Having established the model architecture, we next turned to the data that would serve as the foundation for training and evaluation. We reuse the publicly available corpus released with the study by Aniche et al. [4]. The corpus was created in three steps: repository selection, refactoring extraction, and feature engineering, yielding a large, diverse snapshot of real-world Java development.

*a) Repository selection:* The final set comprises 11,149 projects drawn from three ecosystems:

- Apache Software Foundation: 844 repositories.
- F-Droid (Android): 1,233 applications.
- GitHub: 9,072 highly starred projects.

*b) Refactoring extraction:* RefactoringMiner, which reports 98% recall and 87% precision, scanned every commit history and detected 20 refactoring types spanning class, method, and variable levels (for example Extract Method, Move Class, Rename Variable). In total it identified 2,086,898 refactoring instances from 8.8 million commits. Architectural smells were then linked to canonical refactorings as follows:

- God Class $\rightarrow$ Extract Method.
- Cyclic Dependency $\rightarrow$ Move Class.
- Hub-like Dependency $\rightarrow$ Pull Up Method.

Commits modified at least 50 times without a detected refactoring were sampled as negative instances, producing 1,006,653 non-refactored examples.

*c) Feature engineering:* For every instance we computed three feature groups:

1) Source-code metrics (CK suite, cyclomatic complexity)
2) Process metrics (commit count, bug-fix frequency)
3) Ownership metrics (major author percentage)

All features were normalized to the range [0, 1]. Because refactoring frequencies are uneven, random undersampling balanced minority and majority classes.

*d) Original baseline:* The authors of [4] trained six traditional classifiers (Logistic Regression, Naive Bayes, SVM, Decision Tree, Random Forest, Feed-forward NN) and reported accuracies above 90%. We build on the same balanced dataset but fine-tune transformer models to predict which refactoring best resolves an architectural smell rather than merely forecasting whether any refactoring will occur.

### C. Model fine-tuning and evaluation

With the dataset prepared and the models selected, we proceeded to configure the fine-tuning process through systematic hyperparameter exploration. Both models are fine-tuned as multi-class classifiers using the Hugging Face Trainer API with GPU acceleration (NVIDIA Tesla T4). Input code is tokenised to a maximum length of 512; longer fragments are processed with a sliding-window strategy. Training uses the AdamW optimiser, cross-entropy loss, batch size 16 and

early stopping on validation F1. A random search explores learning rates {1e-5, 2e-5, 5e-5, 7e-5, 8e-5}, batch sizes {8, 16, 32}, and weight-decay schedules. Approximately forty configurations are evaluated; the best validation F1 determines the final hyper-parameters (2e-5 for CodeBERT, 5e-5 for CodeT5). Models are trained for ten epochs with evaluation after each epoch. Performance is reported with accuracy, precision, recall and F1. Ten-fold cross-validation safeguards against project-specific bias. Random seeds are fixed (42) and software versions pinned (Transformers 4.37.2, Datasets 2.16.1, scikit-learn 1.3.2) to ensure reproducibility.

### D. Experiment Execution

We framed refactoring recommendation as a multi-class classification task and fine-tuned CodeBERT and CodeT5 accordingly. Training data were stored in tab-separated files that pair a Java snippet with its refactoring label. Each snippet was tokenised to a maximum length of 512 tokens (RoBERTa tokenizer for CodeBERT, AutoTokenizer for CodeT5). Longer fragments were processed with a sliding-window strategy to avoid losing context. Fine-tuning was performed with the Hugging Face Trainer API on an NVIDIA Tesla T4 GPU. Both models were trained for ten epochs with AdamW, cross-entropy loss, and initial batch size 16.

Approximately 40 hyperparameter combinations were evaluated; the highest validation F1 yielded the final settings (learning rate 2e-5 for CodeBERT, 5e-5 for CodeT5; batch size 16). Evaluation at each epoch reported accuracy, precision, recall and F1. Ten-fold cross-validation mitigated project bias. Figure 1 outlines the workflow from labelled snippets through sliding-window preprocessing to classification.

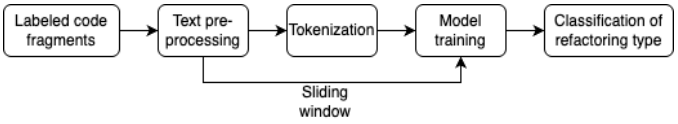



Fig. 1: Pipeline for refactoring-type classification.

Reproducibility was enforced by fixing the random seed to 42, pinning library versions (Transformers 4.37.2, Datasets 2.16.1, scikit-learn 1.3.2) and logging every hyper-parameter. Code, data splits and trained checkpoints are available at https://anonymous.4open.science/r/archsmell_transformers-66F8.

## IV. Results

This section reports the results of the refactoring-type classification experiments. For clarity, findings are organised by research question and supported with the corresponding evaluation metrics (accuracy, precision, recall, F1) and error analyses for both CodeBERT and CodeT5.

### A. $RQ_1$: Are the considered Transformer models capable of correctly predicting the refactoring type required to repair a given architectural smell?

Table I summarizes the performance of the two models. CodeBERT's accuracy rose from 75.5% at epoch 1 to 85.3% at epoch 10, with the macro-averaged F1 following a similar trend (0.76 → 0.85). After epoch 4 the validation loss began to climb while accuracy still improved, suggesting mild overfitting.

TABLE I: Comparison of CodeBERT and CodeT5 at peak performance (Epoch 9).

| Metric | CodeBERT | CodeT5 |
|---|---|---|
| Accuracy | 85.28% | **96.98%** |
| F1-score | 0.8527 | 0.9516 |
| Training Loss | 0.0515 | 0.0052 |
| Validation Loss | 0.9958 | 0.2249 |
| False Positives | ∼266 | 258 |
| False Negatives | ∼266 | 258 |

CodeT5 delivered markedly better and more stable results: the final accuracy and F1 reache 97.0% and 0.95, and the gap between training and validation curves remained small across epochs. Figure 2 plots accuracy, precision, recall and F1 over the ten training epochs; in every metric CodeT5 stays above CodeBERT, confirming the quantitative gains in Table I.

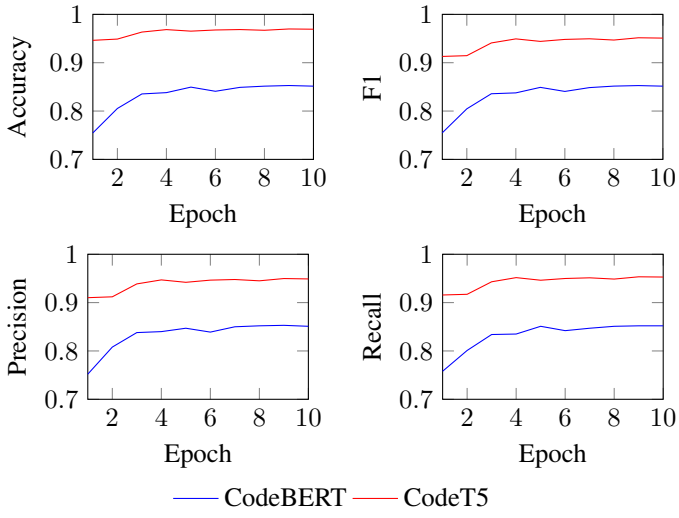


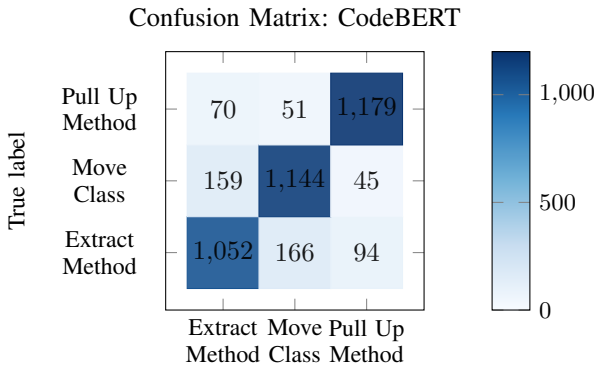

Fig. 2: Performance metrics over 10 epochs for CodeBERT and CodeT5.

**Answer to $RQ_1$:** CodeT5 predicts the correct refactoring with 97% accuracy and 0.95 macro-F1, while CodeBERT reaches 85% accuracy and 0.85 F1, confirming that transformers can reliably map architectural smells to appropriate refactoring operations.

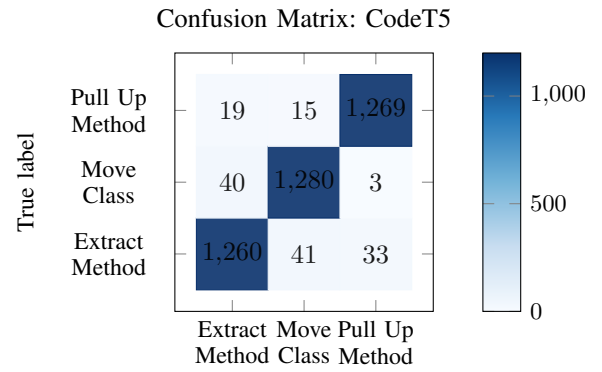
### B. $RQ_2$: How do CodeBERT and CodeT5 differ in effectiveness and computational cost for this task?

The confusion matrices in Figures 3a and 3b confirm that CodeT5 makes fewer mistakes and better separates the three refactoring classes than CodeBERT. Figure 3a shows CodeBERT's confusion matrix. The model classifies Pull Up Method most reliably (precision 87.6 percent, recall 87.2 percent) and performs almost as well on Move Class (84.9

precision, 84.3 recall). Extract Method remains the weak point: although precision is still high at 83.8 percent, recall falls to 79.2 percent because a noticeable fraction of true Extract-Method instances are mistaken for Move Class or Pull Up Method. This suggests CodeBERT struggles to separate refactorings that share similar structural cues. Figure 3b depicts CodeT5's results. Move Class is predicted with 80.2 percent precision and an outstanding 89.8 percent recall, while Pull Up Method reaches 83.3 precision and 87.6 recall. Extract Method remains the hardest case, with recall at 68.7 percent despite 84.9 precision, but CodeT5 still shows fewer cross-class confusions than CodeBERT. Most residual errors involve swapping Extract Method and Move Class, two refactorings that are often applied to similar large or highly coupled classes. Overall, the matrices illustrate CodeT5's superior class separation and explain its higher macro-averaged F1. Table II




(a) CodeBERT predictions




(b) CodeT5 predictions

Fig. 3: Confusion matrices across three refactoring types.

summarizes per-class performance.

**Answer to RQ$_2$:** CodeT5 outperforms CodeBERT (97% vs 85% accuracy; 0.95 vs 0.85 F1) and shows cleaner class separation, yet its encoder-decoder design trains longer and uses more GPU memory. CodeBERT converges faster with lower resource demand but sacrifices predictive quality.

TABLE II: Per-class comparison of CodeBERT and CodeT5 based on confusion matrix analysis.

| Refactoring Type | Model | Precision | Recall | Key Observations |
|---|---|---|---|---|
| Extract Method | CodeBERT | 83.8% | 79.2% | Higher recall than CodeT5 |
| | CodeT5 | 84.9% | 68.7% | More misclassifications as Move Class |
| Move Class | CodeBERT | 84.9% | 84.3% | Balanced performance |
| | CodeT5 | 80.2% | 89.8% | Slightly better recall |
| Pull Up Method | CodeBERT | 87.6% | 87.2% | Best predicted class |
| | CodeT5 | 83.3% | 87.6% | High performance, slightly lower precision |

*C. **RQ$_3$:*** Which architectural smells benefit most from learning-based refactoring recommendations?

Among the three architectural smells examined, God Class, Cyclic Dependency, and Hub-like Dependency, the refactoring predicted most reliably by both models is Pull Up Method, the common remedy for hub-like dependencies. CodeBERT reached 87.6% precision and 87.2% recall for this class, and CodeT5 showed comparable values, indicating that the structural cues of Pull Up Method are distinctive and well captured by the transformers. Extract Method, associated with God Class, proved the hardest to identify. CodeT5 achieved only 68.7% recall for this label, and many true instances were misclassified as Move Class, reflecting the semantic overlap between splitting a large class (Extract Method) and relocating code (Move Class). The present study is limited to these three smells; how transformer-based refactoring performs on other architectural smells remains an open question for future work.

**Answer to RQ$_3$:** Hub-like Dependency benefits the most: both transformers predict its refactoring, Pull Up Method, with the highest precision and recall ($\approx$ 88%), whereas God Class (Extract Method) is hardest and Cyclic Dependency (Move Class) falls in between.

## V. Discussion and Looking Ahead

This section interprets, discusses our empirical findings, and highlights the threats to validity. We first reflect on the comparative performance of CodeBERT and CodeT5 and what the results reveal about transformer architecture choices for refactoring recommendation. We then outline concrete benefits and open questions for tool builders, researchers, and educators. Next, we examine the main threats that could limit the reliability or generalisability of the evidence. Finally, we sketch a research agenda that extends ROSE toward broader smell coverage, cross-language support, interactive repair, and human-centred evaluation.

### A. On the results

CodeT5 achieved 96.9% accuracy and a macro-F1 of 0.95—more than ten points higher than CodeBERT, showing that an encoder–decoder model, even when used only for its encoder, better captures the structural cues distinguishing refactoring types.

This advantage stems from fundamental architectural and pre-training differences: CodeBERT's encoder-only design is optimised for masked-token prediction and code–text alignment, whereas CodeT5's encoder–decoder stack is pre-trained on diverse generation tasks (summarisation, translation, refinement), enabling it to learn longer-range dependencies and subtler structural variations required for multi-class refactoring prediction.

Although CodeBERT converged faster, its rising validation loss after epoch 4 signalled over-fitting, whereas CodeT5 maintained a small generalisation gap and fewer cross-class confusions. Both models predicted Pull Up Method (the fix for hub-like dependencies) most accurately, while Extract Method (the remedy for God Class) proved hardest, often confused with Move Class. This pattern suggests the need for a hierarchical approach that first selects the repair family for a smell, then ranks specific refactorings within that family.

It is worth noting, however, that CodeT5's stronger performance comes at the cost of greater computational demand and longer training and inference times; in resource-constrained or real-time settings, the lighter CodeBERT model may offer a more practical speed–accuracy trade-off. Accordingly, the choice between the two transformers should be guided by deployment goals—high-accuracy offline analysis versus fast online recommendation.

### B. Implications for researchers and practitioners

The results offer tangible benefits for several audiences. Tool builders can embed transformer-based recommenders into existing detectors such as Designite and Arcan, replacing static warnings with concrete, automatically generated fixes. Researchers studying smell evolution can exploit the released two-million-instance corpus to observe how developers repair architectural flaws across projects and time, enabling longitudinal analyses that were previously impractical. Educators likewise gain realistic, data-driven examples: novice architects can examine refactorings suggested by the model and compare them with real-world practice rather than relying solely on textbook illustrations.

### C. Threats to validity

Despite the study's scale and rigour, four threats to validity warrant mention. First, internal validity may be compromised by residual preprocessing or labelling noise, even though we used RefactoringMiner, standard Hugging Face tokenisers, and fixed random seeds. Second, the construct validity is limited by our one-to-one mapping of each smell to a single refactoring; in practice, developers often apply multiple or composite fixes, so some "misclassifications" may be acceptable alternatives. Third, external validity is restricted because the corpus contains only Java projects; assessing transferability to other languages and industrial code bases requires future multi-language replications. Finally, conclusion validity is constrained by reliance on a single large dataset: although we employed ten-fold cross-validation and extensive hyper-parameter search, independent project-out or cross-repository evaluations are needed to confirm robustness before real-world deployment.

### D. Looking ahead

We see four complementary research directions that align with the emerging results and vision focus of this track. First, ROSE should be extended beyond the three architectural smells studied here to a richer catalogue, such as Unstable Interface and Cyclically-Dependent Abstraction, while exploring hierarchical or multi-label predictors that can handle composite fixes. Second, the system can evolve from pure recommendation to interactive repair by linking the classifier to an automatic patch generator (for example, the CodeT5 decoder) and incorporating developer feedback in the loop. Third, cross-language generalisation warrants investigation by fine-tuning multilingual transformers like CodeGemma or StarCoder2 on refactorings mined from languages such as Kotlin, C#, and JavaScript, thereby exposing language-specific biases. Finally, controlled user studies are needed to determine whether the recommendations truly accelerate architecture-maintenance tasks.

Together, these avenues can move transformer-based refactoring support from proof-of-concept to practical, language-agnostic assistance for software architects.

## VI. Conclusion and Future Work

In this paper, we conceived ROSE, a Transformer-based model for recommending software refactoring types based on source code. The study focused on three common refactorings, i.e., *Extract Method*, *Move Class*, and *Pull Up Method*, each of which is related to well-known architectural smells such as God Class and Cyclic Dependency.

The results showed that CodeT5 consistently outperforms CodeBERT in all metrics, achieving a maximum validation accuracy of 96. 98% and a F1 score of 0.9516 in Epoch 9. In contrast, CodeBERT reaches a maximum accuracy of 85.28% and F1-score of 0.8527. Confusion matrix analysis revealed that both models struggled with the *Extract Method* class due to its similarity with *Move Class*, but performed well on *Pull Up Method*. CodeT5 demonstrated stronger generalization capabilities and faster convergence, while CodeBERT showed signs of overfitting in later epochs. These findings confirm the viability of using pre-trained Transformer models–especially encoder–decoder architectures like CodeT5–for automated software refactoring support. Such models can complement traditional static analysis by offering intelligent, data-driven insights into code structure and quality.

We anticipate that there are various future research directions as follows. Some code fragments may involve more than one refactoring; future work could explore models that

support multi-label outputs. Moreover, incorporating attention visualization or saliency maps could help explain why the model chooses a particular refactoring class, increasing trust and usability. Last but not least, larger models like CodeT5-Large or CodeGen could be evaluated for their ability to improve performance on more complex refactorings.

## ACKNOWLEDGMENT

This work is supported by the Swedish Agency for Innovation Systems through the project "Secure: Developing Predictable and Secure IoT for Autonomous Systems" (2023-01899), and by the Key Digital Technologies Joint Undertaking through the project "MATISSE: Model-based engineering of digital twins for early verification and validation of industrial systems" (101140216).

## REFERENCES

[1] J. Al Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Information and Software Technology*, vol. 58, pp. 231–249, 2015.

[2] A. Alazba, H. Aljamaan, and M. R. Alshayeb, "Deep learning approaches for bad smell detection: a systematic literature review," *Empirical Software Engineering*, vol. 28, 2023, corpusID:258591793. [Online]. Available: https://api.semanticscholar.org/CorpusID:258591793

[3] I. Ali, S. S. H. Rizvi, and S. H. Adil, "Enhancing software quality with AI: A transformer-based approach for code smell detection," *Applied Sciences*, vol. 15, no. 8, p. 4559, 2025.

[4] M. Aniche, E. Maziero, R. Durelli, and V. H. S. Durelli, "The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 48, no. 04, pp. 1432–1450, Apr. 2022. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TSE.2020.3021736

[5] F. Arcelli Fontana, M. Camilli, D. Rendina, A. G. Taraboi, and C. Trubiani, "Impact of architectural smells on software performance: an exploratory study," in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 22–31. [Online]. Available: https://doi.org/10.1145/3593434.3593442

[6] F. Arcelli Fontana, I. Pigazzini, R. Roveda, D. Tamburri, M. Zanoni, and E. Di Nitto, "Arcan: A tool for architectural smells detection," in *International Workshops on Software Architecture*, 2017, pp. 282–285, international Workshops on Software Architecture.

[7] U. Azadi, F. Arcelli Fontana, and D. Taibi, "Architectural smells detected by tools: a catalogue proposal," in *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 2019, pp. 88–97.

[8] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Information and Software Technology*, vol. 108, pp. 115–138, 2019.

[9] J. Cordeiro, S. Noei, and Y. Zou, "An empirical study on the code refactoring capability of large language models," *arXiv preprint arXiv:2411.02320*, 2024.

[10] Q. Feng, S. Liu, H. Ji, X. Ma, and P. Liang, "An empirical study of untangling patterns of two-class dependency cycles," 2023. [Online]. Available: https://arxiv.org/abs/2306.10599

[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 *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2020.

[12] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2016, pp. 18:1–18:12.

[13] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[14] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," https://arxiv.org/abs/2009.08366, 2020, arXiv preprint arXiv:2009.08366.

[15] K. Jesse, C. Kuhmuench, and A. Sawant, "Refactorscore: Evaluating refactor prone code," *IEEE Transactions on Software Engineering*, 2023, early Access.

[16] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, 2022, pp. 21 314–21 328.

[17] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, and G. Li, "Codexglue: A machine learning benchmark dataset for code understanding and generation," https://arxiv.org/abs/2102.04664, 2021, arXiv preprint arXiv:2102.04664.

[18] R. Malhotra, B. Jain, and M. Kessentini, "Examining deep learning's capability to spot code smells: a systematic literature review," *Cluster Computing*, vol. 26, pp. 3473–3501, 2023, corpusID:263654376. [Online]. Available: https://api.semanticscholar.org/CorpusID:263654376

[19] T. Mariani and S. R. Vergilio, "A systematic review on search-based refactoring," *Information and Software Technology*, vol. 83, pp. 14–34, 2017.

[20] M. Mohan and D. Greer, "A survey of search-based refactoring for software maintenance," *Journal of Software Engineering Research and Development*, vol. 6, no. 1, pp. 3–55, 2018.

[21] H. Mumtaz, P. Singh, and K. Blincoe, "A systematic mapping study on architectural smells detection," *Journal of Systems and Software*, vol. 173, p. 110885, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302752

[22] P. Naik, S. Nelaballi, V. S. Pusuluri, and D.-K. Kim, "Deep learning-based code refactoring: A review of current knowledge," *SSRN Electronic Journal*, 2023, corpusID:254267544. [Online]. Available: https://api.semanticscholar.org/CorpusID:254267544

[23] A. S. Nyamawe, "Mining commit messages to enhance software refactorings recommendation: A machine learning approach," *Machine Learning with Applications*, vol. 9, p. 100316, 2022.

[24] D. D. Pompeo and M. Tucci, "Multi-objective software architecture refactoring driven by quality attributes," in *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*. IEEE, Mar. 2023, p. 175–178. [Online]. Available: http://dx.doi.org/10.1109/ICSA-C57050.2023.00046

[25] G. Rasool and Z. Arshad, "A review of code smell mining techniques," *Journal of Software: Evolution and Process*, vol. 27, no. 11, pp. 867–895, 2015.

[26] J. A. M. Santos, J. B. Rocha-Junior, L. C. L. Prates, R. S. do Nascimento, M. F. Freitas, and M. G. de Mendonça, "A systematic review on the code smell effect," *Journal of Systems and Software*, vol. 144, pp. 450–477, 2018.

[27] D. Sas, P. Avgeriou, and U. Uyumaz, "On the evolution and impact of architectural smells—an industrial case study," *Empirical Software Engineering*, vol. 27, no. 86, 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10132-7

[28] T. Sharma, P. Mishra, and R. Tiwari, "Designite: A software design quality assessment tool," in *International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities*, 2016, pp. 1–4, international Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities.

[29] S. Singh and S. Kaur, "A systematic literature review: Refactoring for disclosing code smells in object oriented software," *Ain Shams Engineering Journal*, vol. 9, no. 4, pp. 2129–2151, 2018.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[31] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," https://arxiv.org/abs/2109.00859, 2021, arXiv preprint arXiv:2109.00859.

[32] Y. Xiao, X. Zuo, X. Lu, J. S. Dong, X. Cao, and I. Beschastnikh, "Promises and perils of using transformer-based models for se research," *Neural Networks*, vol. 184, p. 107067, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608024009961

[33] M. Zakeri-Nasrabadi, S. Parsa, E. Esmaili, and F. Palomba, "A systematic literature review on the code smells datasets and validation mechanisms," *ACM Computing Surveys*, vol. 55, no. 13s, p. 1–48, Jul. 2023. [Online]. Available: http://dx.doi.org/10.1145/3596908

[34] D. Zhang, S. Song, Y. Zhang, and H. Liu, "Code smell detection research based on pre-training and stacking models," *IEEE Latin America Transactions*, vol. 22, no. 1, pp. 22–30, 2024.

[35] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: A review of current knowledge," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 23, no. 3, pp. 179–202, 2011.

[36] Y. Zhang, C. Ge, H. Liu, and K. Zheng, "Code smell detection based on supervised learning models: A survey," *Neurocomputing*, vol. 565, p. 127014, 2024, dOI:10.1016/j.neucom.2023.127014.