SEMANTIC AND EXECUTION-AWARE EXTRACT METHOD REFACTORING VIA SELF-SUPERVISED LEARNING AND REINFORCEMENT LEARNING-BASED MODEL ALIGNMENT

by

Indranil Palit

Submitted in partial fulfillment of the requirements
for the degree of Master of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
December 2024

© Copyright by Indranil Palit, 2024

To my beloved parents, my younger brother, and my past self, who thought, "Yeah, this sounds like a great idea!"

Table of Contents

List of Tables

List of Figures

Abstract

Software code refactoring is essential for maintaining and improving code quality, yet it remains challenging for practitioners. While modern tools help identify where code needs refactoring, the current implementation techniques often miss meaningful refactoring opportunities. This accumulates technical debt over time, making software increasingly difficult to maintain and evolve. This thesis presents an automated hybrid approach to identify refactoring candidates and generate refactored code leveraging language models and reinforcement learning.

The first major contribution of the thesis addresses the shortcomings of automatic refactoring candidate identification by training machine learning classifiers with rich code semantics. Unlike traditional approaches that rely on metrics and commit messages, we've developed a self-supervised learning approach to identify negative samples utilizing state-of-the-art GraphCodeBERT embeddings. This approach achieves a 30% improvement in F1 score compared to existing metric-based techniques to identify extract method refactoring candidates automatically.

Our second contribution introduces a novel approach to automated code refactoring using reinforcement learning, with a specific focus on extract method refactoring. While recent advances in large language models have shown promise for code transformation, traditional supervised learning approaches often fail to produce reliable results. These models typically struggle with maintaining code integrity because they treat code generation similar to text generation, overlooking crucial aspects like compilability and functional correctness. To address this limitation, we develop a method that fine-tunes state-of-the-art pre-trained code language models (e.g., CodeT5) with Proximal Policy Optimization (PPO), creating a code-aware transformation framework. Our approach uses carefully designed reward signals based on successful compilation and adherence to established refactoring guidelines, moving beyond simple text-based metrics. When tested against conventional supervised learning methods, our system demonstrates significant improvements in quality and quantity.

Together, these advances mark a significant step toward automated, reliable code refactoring that can effectively reduce software practitioner's burden while maintaining high software quality standards. The success of our approach demonstrates how modern machine learning and deep learning techniques when carefully adapted to understand code semantics, can transform traditional software engineering practices.

Acknowledgements

Firstly, I can't thank Dr. Tushar Sharma enough. He's been an incredible supervisor since before I even joined Dalhousie. The trust he placed in me was amazing! Beyond research guidance, he helped with teaching opportunities and external projects. What makes him special is how approachable he is - whether it's academic or personal issues, his door was always open. While I wasn't always a perfect student and made mistakes along the way, he helped me learn and grow from each experience. His endless encouragement, thoughtful guidance, and understanding nature made this thesis possible. Tushar, you're the best mentor I could've asked for - thank you for everything!

To the members of our Software Maintenance and Analytics Research Team (SMART) lab - Mootez Saad, Saurabhsingh Rajput and Gautam Shetty, I consider myself to be extremely fortunate to have met you all and got the opportunity to work alongside such great minds. I am also grateful for all the interactive lab meetings we had, from brainstorming research ideas to dreaming up (hilariously failed) startup plans. Those late night discussions about everything under the sun - football, politics, food recipes, and those endless jokes, are memories I'll always cherish. I would like to specially thank my exceptionally talented and hard working friend and peer, Mootez, for time and again helping me with my research providing his insightful feedback and inputs. These suggestions had a significant impact in polishing my work. Working with him was truly inspirational. I would also like to thank the previous members and interns of our lab. It is an honor to become an alumni of this lab.

To Saurabh and Gautam, this section doesn't have sufficient space to accommodate everything that they have done for me. We three have become brothers since day one. Every memory in Halifax involves these two - from last minute assignment rushes and technical discussions to exploring the city together, watching football matches, and impromptu trips. Saurabh's laid-back attitude and reassuring presence with deep technical knowledge helped me navigate through challenging times. Gautam's wisdom and thoughtful insights, both on and off work discussions have been invaluable.

I am also indebted to Gautam for his valuable contribution as an author in my first research paper.

I am deeply thankful to all the members of the Intelligent Automation in Software EngineeRing (RAISE) lab - Mehil B Shah, Asif Samir, Usmi Mukherjee, Sigma Jahan and Riasat Mahbub. I would like to acknowledge and appreciate my very good friends Usmi, Mehil and Sigma for supporting me and cheering me both professionally and personally. Mehil is very vocal and I admire his innocence and authentic presence. Sigma is a multi-tasker and I really think she has invisible octopus hands. Finally, I would like to thank Usmi for being there as a friend and a strong support system throughout the journey. She is one of the best people I ever met. Also, she is the only friend from my hometown, Kolkata. Celebrating Durga Puja as *probashi bangalis* (expatriate bengalis) will always be a great experience.

I would like to extend my sincere gratitude towards Dalhousie University and the Faculty of Computer Science for providing me with this splendid learning opportunity. I would like to thank Dr. Michael McAllister for promptly resolving a major administrative issue I was facing. I would also like to thank the department for awarding me with graduate bursaries twice. I am very fortunate to have received funding support from `NRC IRAP`, `Lab2Market` and `Mitacs Accelerate`. The internships and projects enabled by this funding provided valuable opportunities to apply my research in practical, real-world settings. The defense

This graduate journey would not have been possible without the constant support and encouragement from my incredible parents, brother and other close family members, who have stood by me every step of the way. I am deeply grateful for their belief in me and the strength they have given me throughout this journey. I would also like to take a moment to thank all my friends back home, especially Jessica Banik and Swagnik Banerjee, whose friendship and support have been invaluable during this transformative phase of my life.

Lastly, I sincerely thank everyone who supported me throughout this research journey. Their encouragement, advice, and assistance have been invaluable, and I am truly grateful.

Chapter 1

Introduction

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.
— Martin Fowler

This opening chapter frames our research presented in this thesis, by establishing its context, identifying the key challenges in software refactoring, describing our proposed methodology, and highlighting the main contributions of this work.

1.1 Code quality

Software quality is about more than just ticking boxes on a requirements list. While we can measure it by how well the code matches technical specifications, the real test is whether it actually helps people do what they need to do. At its core, quality software should both work as designed and make users' lives easier [98]. Poor quality choices tend to have lasting consequences, often creating technical debt that impacts projects long after release [88].

In software engineering, quality encompasses two fundamental dimensions: functional and structural [10]. Functional quality measures how effectively software meets its specified design requirements and fulfills its intended purpose, essentially determining if we built the right software. Structural quality, on the other hand, focuses on the non-functional attributes such as maintainability, reliability, and robustness, determining if we built the software right [64]. Together, these dimensions ensure that software not only provides the right features but also performs efficiently and remains sustainable throughout its lifecycle.

Maintainability encompasses key attributes such as modularity, understandability, and testability that determine how easily software can be modified and managed. Studies have shown that poor maintainability practices lead to 40-70% of the total software lifecycle costs [87].

Code maintainability problems often reveal themselves through what software practitioners call “code smells” - patterns that signal potential design flaws [27]. Just as unpleasant odors can signal health hazards in our environment, code smells warn of deeper implementation [27], design [100] or architectural [29] issues. Common examples include duplicated code blocks, overly complex methods, and tightly coupled components. While these patterns don’t necessarily indicate immediate bugs, they make the codebase increasingly difficult to understand, maintain, and modify. Over time, these issues compound, leading to higher development costs and reduced team productivity, particularly when teams need to add new features or fix existing problems.

1.2 Problem statement

One of the code smells, the *long method* smell, characterized by methods that are excessively long and complex, is particularly prevalent in Object-Oriented Programming (OOP) [28, 94]. Recent systematic reviews and empirical studies have consistently ranked the long method smell among the top five most frequently occurring and problematic code smells [1, 55].

The primary remedy for addressing the long method smell is *Extract Method* refactoring [27]. This technique involves identifying cohesive blocks of statements within a long method and extracting them into a separate, well-named method.

However, the current practice of *extract method* refactoring presents several challenges:

- Manual Identification: Developers must first identify methods that require refactoring and then locate specific code blocks suitable for extraction, a process that becomes increasingly complex and time-consuming as codebase size grows.

- Error-Prone Implementation: The manual extraction process can inadvertently introduce new bugs or even create new code smells, particularly when dealing with complex method dependencies and variable scoping.

- Automated Tool Support: The existing tools and IDE plugins that assist software practitioners to perform extract method refactoring are not fully automated. The user has to specify which block of code needs needs the refactoring and then the tool simply extracts that block.

While some approaches have been proposed to address the long method smell [66, 106, 12], these solutions are not fully automated and still require significant manual intervention to complete the refactoring process. The lack of a comprehensive, automated solution for both identifying and executing *extract method* refactoring opportunities represents a significant gap in modern software maintenance tools and practices.

1.3 Motivating example

To illustrate the challenges and importance of automated extract method refactoring, consider the following real-world code example from Netflix's Eureka project at commit 756bcd9 [1], a service registry for microservices architectures:

[1] Netflix Eureka DiscoveryClient.java

Listing 1 Extract Method Refactoring Candidate

```java
private synchronized void updateInstanceRemoteStatus() {
    InstanceInfo.InstanceStatus currentRemoteInstanceStatus = null;
    if (instanceInfo.getAppName() != null) {
        Application app = getApplication(instanceInfo.getAppName());
        if (app != null) {
            InstanceInfo remoteInstanceInfo =
            app.getByInstanceId(instanceInfo.getId());
            if (remoteInstanceInfo != null) {
                currentRemoteInstanceStatus = remoteInstanceInfo.getStatus();
            }
        }
    }
    if (currentRemoteInstanceStatus == null) {
        currentRemoteInstanceStatus = InstanceInfo.InstanceStatus.UNKNOWN;
    }
    if (lastRemoteInstanceStatus != currentRemoteInstanceStatus) {
        try {
            if (eventBus != null) {
                StatusChangeEvent event = new
                StatusChangeEvent(lastRemoteInstanceStatus,
                        currentRemoteInstanceStatus);
                eventBus.publish(event);
            }
        } finally {
            lastRemoteInstanceStatus = currentRemoteInstanceStatus;
        }
    }
  }
```

This method combines status retrieval and event publishing logic with complex control flow and exception handling. The refactored version separates these concerns more effectively:

Listing 2 Code after Extract Method Refactoring

```java
private synchronized void updateInstanceRemoteStatus() {
    InstanceInfo.InstanceStatus currentRemoteInstanceStatus = null;
    if (instanceInfo.getAppName() != null) {
        Application app = getApplication(instanceInfo.getAppName());
        if (app != null) {
            InstanceInfo remoteInstanceInfo =
            app.getByInstanceId(instanceInfo.getId());
            if (remoteInstanceInfo != null) {
                currentRemoteInstanceStatus = remoteInstanceInfo.getStatus();
            }
        }
    }
    if (currentRemoteInstanceStatus == null) {
        currentRemoteInstanceStatus = InstanceInfo.InstanceStatus.UNKNOWN;
    }
    onRemoteStatusChanged(lastRemoteInstanceStatus, currentRemoteInstanceStatus);
    lastRemoteInstanceStatus = currentRemoteInstanceStatus;
}

protected void onRemoteStatusChanged(InstanceInfo.InstanceStatus oldStatus,
InstanceInfo.InstanceStatus newStatus) {
    // Publish event if an EventBus is available
    if (eventBus != null) {
        StatusChangeEvent event = new StatusChangeEvent(oldStatus, newStatus);
        eventBus.publish(event);
    }
}
```

The refactoring improves the code by separating concerns, simplifying error handling, and enhancing testability. However, identifying such opportunities in large codebases requires careful analysis of context, variable scope, and behavior preservation. This example demonstrates the need for an automated approach to detect and perform such refactorings consistently across large-scale systems.

1.4 Proposed Solution and Contribution

1.4.1 Proposed solution

We present a novel, fully automated two-phased approach to address the challenges of extract method refactoring.

Phase 1—Refactoring Candidate Identification: We develop an innovative approach to identify potential candidates for *extract method* refactoring by generating source code embeddings and advanced machine learning classifiers. This phase focuses on identifying candidates suitable for *extract method* refactoring.

Phase 2—Automated Refactoring Generation: We implement an intelligent refactoring system using Large Language Models (LLM), enhanced through fine-tuning and Reinforcement Learning (RL) optimization. This approach focuses on to generate refactored code that maintains both syntactic correctness and semantic equivalence with the original implementation.

1.4.2 Key contributions

This thesis makes the following contributions to the field of automated software refactoring:

- **Novel Sample Identification Framework:** We introduce a systematic mechanism for identifying and classifying positive and negative samples for *extract method* refactoring, addressing a fundamental gap in existing approaches.

- **Comprehensive Benchmark Dataset:** We develop a large-scale dataset comprising $55,430$ labeled samples (both positive and negative), establishing a robust benchmark for evaluating automated refactoring candidate identification approaches.

- **Advanced Code Representation:** We propose an Autoencoder-based approach that leverages GraphCodeBERT for extracting latent features, enabling more effective method representation for binary classification tasks.

- **Supervised Fine-tuning Framework:** We develop and evaluate supervised fine-tuned models specifically designed for automatic *extract method* refactoring, eliminating the need for manual code block selection and addressing key limitations of existing approaches.

- **Hybrid Learning Architecture:** We present a novel hybrid approach that combines supervised fine-tuning with reinforcement learning optimization, specifically engineered for extract method refactoring. The effectiveness of this approach is validated through both quantitative metrics and qualitative analysis.
- **Open-source Tool and Dataset:** We contribute a comprehensive tool for analyzing Java repositories on GitHub to generate extract method refactoring datasets with detailed metadata. Both the tool and the resulting dataset are made publicly available in Appendix A.1 to facilitate replication studies and future research extensions.

1.5 Related Publication

Parts of this research work has been either accepted at or submitted to the following conferences.

- Palit, Indranil, Gautam Shetty, Hera Arif, and Tushar Sharma. "Automatic refactoring candidate identification leveraging effective code representation." In 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 369-374. IEEE, 2023.
- Palit, Indranil and Tushar Sharma. "Reinforcement Learning vs Supervised Learning: A tug of war to generate refactored code accurately" Proceedings of the ACM on Software Engineering 1, no. FSE (2025) (*submitted*)

1.6 Outline of the Thesis

This thesis presents an end-to-end solution for automating *extract method* refactoring through two interconnected studies, organized into five chapters:

- **Chapter 2** Introduces fundamental concepts in software refactoring, code embeddings, transformer architectures, and reinforcement learning that form the foundation for our technical approaches.
- **Chapter 3** Presents our first contribution: an innovative approach using code embeddings and machine learning to automatically identify potential *extract method* refactoring candidates.
- **Chapter 4** Describes our second contribution: a novel reinforcement learning-based framework that automatically transforms identified candidates into properly refactored methods.
- **Chapter 5** Synthesizes our findings and outlines promising directions for future research in automated code refactoring.

Chapter 2

Background and Related work

Research is to see what everybody else has seen, and think what nobody has thought.
— Dr. Albert Szent-Gyorgyi

This chapter has two main sections. The first section provides the background concepts needed to understand our research. The second section reviews existing research in this field, examining prior work that shapes the basis of our study.

2.1 Context

2.1.1 Extract method refactoring

Extract Method refactoring, first formalized by Fowler [27], is a fundamental technique for improving code maintainability by decomposing large methods into smaller, more focused units. This transformation involves identifying cohesive code fragments and extracting them into new methods while preserving program behavior [70]. Recent studies by Silva *et al.* [96] and Tsantalis *et al.* [108] have shown that Extract Method is among the most frequently applied refactorings in practice, accounting for approximately 15-25% of all refactoring operations in large-scale systems.

The primary motivation for Extract Method refactoring is to address the"Long Method" code smell, where methods become long, complex or incohesive, violating the single responsibility principle [65]. The process involves identifying a coherent fragment of code, extracting it into a new method with an appropriate name, and replacing the original fragment with a call to the newly created method. This transformation must preserve program semantics while managing variable scope, parameter passing, and return values [47]. Studies by Murphy-Hill et al. [70] have shown that while developers frequently perform this refactoring manually, automated tool support remains challenging due to the complexity of ensuring behavioral preservation and identifying optimal extraction points.

2.1.2 Embeddings

Code embeddings are dense vector representations that capture semantic properties of source code in a continuous space. Alon *et al.* [6] introduced Code2Vec, demonstrating how path-based attention over abstract syntax trees can create meaningful code representations. This was further advanced by Feng *et al.*. [25] with CodeBERT, which combines both structural and textual features of code. These embedding techniques have revolutionized code analysis by enabling deep learning models to better understand and process source code semantics.

Recent advancements in code embeddings have focused on incorporating multiple views of code, such as data flow, control flow, and natural language documentation. Guo et al. [33] demonstrated that unified representations combining these different aspects significantly improve performance on some downstream tasks. The effectiveness of these embeddings has been particularly evident in tasks such as code similarity detection, bug localization, and automated program repair [131]. Furthermore, studies by Kanade *et al.* [41] have shown that pre-training on large code corpora can create embeddings that capture rich semantic relationships between different programming concepts and patterns.

2.1.3 Transformers

The Transformer architecture, introduced by Vaswani *et al.* [115], revolutionized sequence processing through its self-attention mechanism. In the context of code processing, Ahmad *et al.* [2] demonstrated how Transformers can effectively capture long-range dependencies in source code, while Hellendoorn *et al.* [35] showed their effectiveness in modeling code structure through global attention patterns. The architecture's ability to process sequences in parallel while maintaining contextual relationships has made it particularly suitable for code analysis tasks.

The key innovation of Transformers lies in their self-attention mechanism, which allows the model to weigh the importance of different parts of the input sequence dynamically. For code processing, this is particularly valuable as it enables the model to capture relationships between distant parts of the code, such as variable declarations and their uses, or function calls and their definitions [118]. Recent adaptations of the Transformer architecture for code, such as CodeT5 [121], have introduced specialized attention patterns that better align with code structure, including syntax-aware attention and hierarchical representations that respect code's abstract syntax tree structure.

2.1.4 Supervised fine-funing of large language models

Supervised fine-tuning is an add-on training for adapting pre-trained large language models (LLMs), such as CodeT5 [121], to specialized tasks. This adaptation is achieved by training the models on domain-specific datasets, which is particularly important for enhancing their performance in tasks such as extract method refactoring. In this context, we focus on two predominant model architectures: *encoder-decoder* and *decoder-only* models.

Encoder-decoder models consist of two main components. The encoder processes the input sequence (*i.e.,* source code, in our case) to create a context-rich representation, which the decoder then uses to generate the output sequence (refactored code with extracted method, in our case). This architecture is particularly useful when the input is a code snippet, and the output is the corresponding refactored version. The fine-tuning objective for encoder-decoder models aims to maximize the conditional probability of the correct output sequence given an input sequence. A technique called teacher forcing is employed, where the correct output token from the previous time step is fed as input to the next step.

Decoder-only models, such as those used in GPT-like architectures [82], operate differently. They generate each token of the output sequence directly, conditioned on all previous tokens and the input sequence, without a separate encoding phase. The training process involves presenting the combined sequence of the input code and the refactored code to the model, typically separated by a special token, *e.g.,* [SEP].

For both architectures, the loss function commonly used is the cross-entropy loss, calculated over the output sequence tokens. This loss function helps the model learn to predict the correct tokens in the output sequence.

2.1.5 Reinforcement learning for sequence generation

Reinforcement Learning (RL) is a branch of machine learning focused on training agents to take actions in an environment to maximize some notion of cumulative reward often involving a series of decisions [91]. It uses a model known as the Markov Decision Process (MDP) [81], which deals with decision-making where each action is determined by steps, and outcomes are influenced by randomness. In RL, an agent (*i.e.,* an autonomous entity that takes action in the given environment) improves its decisions through trial-and-error interactions with its environment, learning from the rewards it receives based on its actions. The agent's decision-making strategy is known as the *policy*, which determines the next action to take given the current situation or *state*. The *state* represents the current context or input on which the agent bases its decisions.

In the context of language models, RL can be employed as a training mechanism. Here, the language model serves as the *policy*, and the current text sequence is the *state*. The model generates an action, the next word or token, altering the state into a new text sequence. The quality of the completed text sequence determines the reward, assessed either by human judgment or a trained reward model based on human preferences.

Formally, MDP can be defined as follows.

State Space: The state space S is the set of all possible language representations up to a maximum length L:

$$S = \{s \mid s \in \mathbb{R}^d \times l, l \leq L\} \tag{2.1}$$

where d is the dimensionality of the token embeddings and l is the current sequence length.

Action Space: The action of the model (actor) is to generate the next token based on the previously generated tokens and the input data. The action space is the vocabulary of the language model, representing all possible tokens:

$$A = \{1, 2, \ldots, V\} \tag{2.2}$$

where V is the vocabulary size.

State Transition Function: The state transition function P describes how the environment moves from one state to another given an action. In our sequence generation task, this transition is deterministic:

$$P(s_{t+1} \mid s_t, a_t) = \begin{cases} 1 & \text{if } s_{t+1} = [s_t; e(a_t)], \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

where $e(a_t)$ is the embedding of the chosen token.

Policy: The policy is represented by a language model that produces a probability distribution over the next token given the current state:

$$\pi_\theta(a \mid s) = \text{LM}_\theta(a \mid s) \tag{2.4}$$

where LM_θ is the language model with parameters θ.

Value Function: The value function estimates the expected cumulative reward from a given state under a particular policy:

$$V_\phi(s) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t R_t \mid S_0 = s\right] \tag{2.5}$$

where ϕ are the parameters of the value head and γ is the discount factor.

Reward

In sequence modeling, the reward quantifies the quality or desirability of generated sequences. It measures how well the generated sequence aligns with desired characteristics, such as fluency, coherence, or task-specific objectives.

Reward Function

The reward function R can be expressed as:

$$R(s, a) = \lambda_1 r_{\text{task}}(s, a) + \lambda_2 r_{\text{seq}}(s, a) \tag{2.6}$$

where r_{task} represents task-specific evaluation metrics, r_{seq} captures sequence quality measures, and λ_1, λ_2 are weighting coefficients balancing different reward components..

In the software engineering domain, RL has been used for code completion tasks [95, 57] and code summarization [116]. They all use actor-critic methods to train the language models for specific downstream tasks. The actor is the policy model, the main language model pre-trained or fine-tuned on code data and the critic is another component that evaluates the output generated by the actor and provides a reward signal. Based on this architecture, we formulate our problem as follows.

2.1.6 Solution framework

In this second part of this thesis, we focus on aligning a fine-tuned large language model for extract method refactoring generation using Proximal Policy Optimization (PPO) [86]—a popular actor-critic reinforcement learning method. This alignment process involves several key components: the actor, the critic, rewards, the value function, and KL divergence.

The *actor* in our setting is the language model itself, which generates sequences of code such as extracting methods from code snippets. It takes the current code as input and outputs a refactored version with extracted methods. The *critic* is a separate component that evaluates the quality of the refactoring produced by the actor, providing a score or reward that reflects how well the generated refactoring meets the desired criteria, such as syntactically and semantically accurate refactored code.

A *reward* is a numerical score assigned to each generated refactoring, indicating its quality. Higher rewards are given for refactorings that improve code properties, while lower rewards indicate poor refactoring outcomes, such as introduction of errors. These rewards guide the actor in learning to generate more desirable refactorings over time.

The *value function* estimates the expected reward from a given state or step in the sequence generation process. It predicts how good the current refactoring is, considering future rewards. In practice, the value function is represented by a separate neural network head, called a *value head*, which outputs a scalar value for each input state, estimating the expected cumulative reward, denoted as

$$V(s) = \mathbb{E}\left[R \mid s\right], where\ R\ isthe\ total\ reward \tag{2.7}$$

*Proximal Policy Optimization (*PPO*)* is the algorithm used to train the actor model. PPO optimizes the model's parameters by adjusting its behavior in small, controlled steps, ensuring that changes are not too drastic. This balance between exploration (trying new refactoring strategies) and stability (maintaining effective behaviors) helps the model learn efficiently without losing its learned knowledge.

The *objective function* in PPO aims to maximize the expected reward while ensuring that updates to the policy (the actor's behavior) are not excessively large. The PPO objective can be formulated as:

$$L^{\text{PPO}}(\theta) = \mathbb{E}\left[\min\left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right],$$

where θ represents the model parameters, $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the ratio of the new policy to the old policy, $\hat{A}_t$ is the advantage estimate, and ϵ is a small constant controlling the clipping range.

The *PPO loss* combines the clipped objective function with the value function loss and an entropy bonus. The complete loss function can be expressed as:

$$L(\theta) = L^{\text{PPO}}(\theta) - c_1 \cdot \text{Value Loss}(\theta) + c_2 \cdot \text{Entropy Bonus}(\theta),$$

where the *Value Loss* measures the accuracy of the critic's value estimates, and the *Entropy Bonus* encourages exploration by preventing the actor from becoming overly deterministic.

KL Divergence (Kullback-Leibler divergence) measures the difference between the old policy $\pi_{\theta_{old}}$ and the updated policy π_θ, ensuring that updates to the policy do not deviate excessively from the original behavior. It is calculated as:

$$\text{KL}(\pi_{\theta_{old}} \parallel \pi_\theta) = \sum_x \pi_{\theta_{old}}(x) \log\left(\frac{\pi_{\theta_{old}}(x)}{\pi_\theta(x)}\right), \tag{2.8}$$

where $\pi_{\theta_{old}}(x)$, $\pi_\theta(x)$ represent the probability distributions over the possible code refactoring actions that the model can take at a given step.

In summary, the actor generates refactoring suggestions, and the critic evaluates them using static non differential rewards that provide feedback on their quality. PPO optimizes the model's behavior gradually, guided by the value function, the objective function, and loss components, while KL divergence ensures that changes remain within reasonable limits. This framework enables the fine-tuning of the language models to produce high-quality code refactorings over time.

2.2 Related Work

2.2.1 Refactoring candidate identification using traditional techniques

Software developers can manually decide *what* to refactor according to their intuition and past experiences [4]. Often, they use automated tools to calculate code quality metrics and code smells to identify refactoring candidates [68, 4]. Another method to identify refactoring candidates is to define a set of preconditions or compliance rules. If a code did not follow these rules, it was considered a candidate for refactoring. Studies by Bois *et al.* [21] and Kataoka *et al.* [44] used such compliance rules.

Another technique considers creating clustering algorithms to identify if code needs refactoring. Czibula *et al.* [17] and Serban *et al.* [89], created clusters based on the distance between methods and attributes within and outside of classes to identify numerous refactoring candidates. Similarly, Bavota *et al.* [11] suggest a graph-based approach that uses weighted graphs instead of abstract syntax trees to identify methods that can be extracted. Finally, Tsantalis *et al.* [105] used code slices to identify *extract method* candidates.

2.2.2 Detecting refactoring with Machine Learning techniques

Many studies have explored ways to identify refactoring candidates automatically using machine learning. Typically, such studies use source code metrics or commit messages to train a model. For example, Aniche *et al.* [7] predict 20 kinds of refactorings at the method, class, or variable level. They use a large number of code, process, and ownership metrics to train six supervised machine learning algorithms. The study reports that *Random Forest* model performs the best among the compared models. Gerling [30] conducted an empirical study as an extension of Aniche *et al.*'s study. They focused on improving the data collection process in Aniche *et al.*'s study to create a high quality large-scale refactoring dataset.

Similarly, Van Der Leij *et al.* [112] analyze five machine learning models to predict *extract method* refactoring and compare the results with industry experts. They collect 61 code metrics and analyze *Random Forest*, *Decision Tree*, *Logistic Regression*, Linear SVM, and Gaussian *Naive Bayes* algorithms. They found *Random Forest* as the best performing model. Kumar *et al.* [52] perform a study to predict method-level refactoring and analyze 10 machine learning classifiers.

Sagar *et al.* [85] considers the problem of refactoring candidate prediction as a multi-class classification problem. Their study uses source code quality metrics and commit messages as features to predict six method-level refactorings. They compare two machine learning models: a text-based model that analyses keywords from commit messages and a source code-based model that analyses 64 code quality metrics. Kurbatova *et al.* [53] use code embeddings generated from Code2Vec [6] to train their machine learning model to predict the *move method* refactoring.

2.2.3 Automated Refactoring

Many studies have explored automated refactoring candidate identification using machine learning techniques. Typically, these studies use source code metrics or commit messages to train models. Aniche *et al.* [7] predict 20 kinds of refactorings at method, class, or variable levels using code, process, and ownership metrics, with Random Forest performing best among six algorithms. Gerling [30] extended this work by improving the data collection process to create a high-quality, large-scale refactoring dataset. Van Der Leij *et al.* [113] analyze five machine learning models to predict Extract Method refactoring, comparing results with industry experts. Using 61 code metrics, they also found Random Forest to be the best performing model. Kumar *et al.* [52] studied method-level refactoring prediction, analyzing 10 machine learning classifiers. Sagar *et al.* [85] approaches refactoring candidate prediction as a multi-class classification problem, using both source code quality metrics and commit messages as features to predict six method-level refactorings. They compare text-based and source code-based models. Kurbatova *et al.* [53] employ code embeddings generated from Code2Vec [6] to train their model for Move Method refactoring prediction.

In the domain of automated code refactoring, researchers have developed a variety of specialized tools and approaches. CeDAR [102], an Eclipse plugin, focuses on identifying and eliminating duplicate code. JDeodorant [104, 67] detects code smells and proposes refactoring strategies. Fokaefs *et al.*[26] extended JDeodorant's capabilities to prioritize and implement class extraction refactorings. SOMOMOTO[128] facilitates move method refactoring and code modularization. While these rule-based methods have made significant contributions, they face limitations in capturing semantic information during refactoring. Moreover, they often require manual intervention from developers to identify and select code blocks for refactoring. To address these challenges, recent research has explored the application of deep learning and large language models (LLMs) for automated code refactoring.

Szalontai *et al.* [101] developed a deep learning method for refactoring source code, initially designed for the Erlang programming language. Their approach comprises a localizer and a refactoring component, enabling the identification and transformation of non-idiomatic code patterns into idiomatic counterparts. Tufano *et al.* [111] conducted a quantitative investigation into the potential of Neural Machine Translation (NMT) models for automatically applying code changes implemented during pull requests. Their approach leverages NMT to translate code components from their pre-pull request state to their post-pull request state, effectively simulating developer-implemented changes. To facilitate the rename refactoring process and reduce cognitive load on developers, Liu *et al.* [61] proposed RefBERT, a two-stage pre-trained framework based on the BERT architecture. RefBERT is designed to automatically suggest meaningful variable names, addressing a challenging aspect of code refactoring.

Current automated refactoring tools lack semantic understanding and require manual intervention. To address this, we propose a hybrid approach combining supervised fine-tuning with reinforcement learning, enhancing the accuracy and completeness of extract method refactoring. This is the first study to apply deep reinforcement learning for this task, advancing automated refactoring tools.

2.2.4 Reinforcement learning in software engineering

Sequence modeling has emerged as a fundamental paradigm for addressing a wide array of software engineering challenges. In recent years, researchers have explored the application of deep reinforcement learning (DRL) techniques to mitigate exposure bias in supervised fine-tuned models for sequence generation tasks [83, 45]. Notably, Ranzato *et al.* [83] pioneered the use of established metrics such as BLEU and ROUGE as reward signals in DRL algorithms to optimize network parameters in machine translation, effectively addressing exposure bias. The intersection of DRL and sequence modeling has led to innovative frameworks, such as the one proposed by Chen *et al.* [14], which reconceptualizes reinforcement learning problems as sequence modeling tasks. This approach has paved the way for novel applications in various domains.

In the realm of software engineering, DRL methods have gained traction, particularly in code completion and summarization tasks. Wang *et al.*[117] leveraged compiler feedback as a reward signal to enhance the quality of language model-generated code. Le*et al.*[56] introduced CodeRL, a framework that integrates RL with unit test signals to fine-tune program synthesis models. Shojaee *et al.* [95] conducted comprehensive research, proposing a framework for fine-tuning code language models using DRL and execution signals as rewards. Recent advancements in this field include IRCOCO by Li *et al.*[57], which employs immediate rewards to fine-tune language models for code completion tasks. Wang*et al.*[119] developed RLCoder, combining DRL with Retrieval-Augmented Generation (RAG) pipelines for repository-level code completion. Furthermore, Nichols*et al.* [73] demonstrated the potential of DRL in generating efficient parallel code, expanding the application of these techniques to performance optimization.

To our knowledge, large language models have not been specifically trained or aligned for extract method refactoring. Our approach, which combines supervised fine-tuning with PPO alignment, is a first in this domain. This novel methodology produces accurate refactored methods, marking a significant advancement in the field.

Chapter 3

Automatic Refactoring Candidate Identification

The function of good software is to make the complex appear to be simple.
— Grady Booch

In this chapter we introduce a novel solution for automated refactoring candidate detection that combines self-supervised learning with pre-trained language models. Our approach moves beyond conventional metric-based methods to capture deeper code semantics, demonstrating substantial improvements in accuracy and practical applicability.

3.1 Introduction

Refactoring is a process and a set of techniques to improve source code's structure and quality without impacting the behavior and functionality of the software, thus making the software more maintainable [75, 27]. It is a widely used technique among software developers as it improves code readability, testability, flexibility, and adaptability to introduce changes to meet new requirements [68].

One of the critical questions that a developer answers during software development is whether a source code entity (*e.g.,* a method or a class) requires refactoring and if yes, what is the most appropriate refactoring in the context. Practitioners decide *what* to refactor according to their intuition and experience. They can also use automated tools to calculate code quality metrics and code smells [94]. However, metrics and smells focus on the *problem* aspect and provide little help to decide whether and what refactoring technique must be applied to remove the smell or improve the metrics.

To overcome this challenge, many studies propose techniques to predict refactoring candidates by analyzing source code properties [7, 42, 53]. However, existing efforts in this direction suffers from several deficiencies. Currently, the state-of-the-art research in detecting refactoring candidates, such as Aniche *et al.* [7], follows a metric-based approach, *i.e.,* it collects code quality and process metrics, and trains a machine learning model using the collected metrics as features. Similarly, Xu *et al.* [123], use variable accesses in addition to code quality metrics. These approaches fail to capture the hidden contextual and syntactical characteristics of code that might contribute to better refactoring candidate identification. To overcome the issue, researchers have used code embeddings [53, 42] extracted from Code2Vec [6]. However, existing studies in this domain do not capture rich contextual and syntactical characteristics of code [43]; such information could significantly improve the performance of the identified refactoring candidates. Secondly, existing studies lack an appropriate mechanism to define and identify negative samples for their dataset in this context. A code snippet is considered a negative sample if it is not a candidate for the specific refactoring. Typically, studies use tools such as RefactoringMiner [109, 107] to identify positive code samples. However, to identify negative samples, researchers define unsound rule-based heuristics resulting in a low-quality noisy dataset [103]. Finally, most of the previous research in this field fails to consider the real-world ratio of positive and negative samples while evaluating the predictive models. Ignoring this guideline results in a model that works well in an experimental study but performs poorly when deployed in a real-world context [92, 20].

In this thesis, we address the aforementioned deficiencies. We present an automated Deep Learning (DL)-based technique to identify candidates for *extract method* refactoring. The *extract method* refactoring isolates a code block from a larger method and generates a new method based on the extracted code snippet [27]. We kept our focus on *extract method* because it is one of the most commonly used refactoring [71]. We create our dataset from open-source Java repositories and prepare an effective code representation capturing syntactic and semantic properties of methods by combining GraphCodeBERT [34] and Autoencoder [60]. The representation is then used to identify *extract method* refactoring candidates.

Contributions of the study: We propose a novel mechanism to properly identify positive and negative samples for *extract method* refactoring. The mechanism helps us create a dataset containing $55,430$ positive and negative samples that serves as

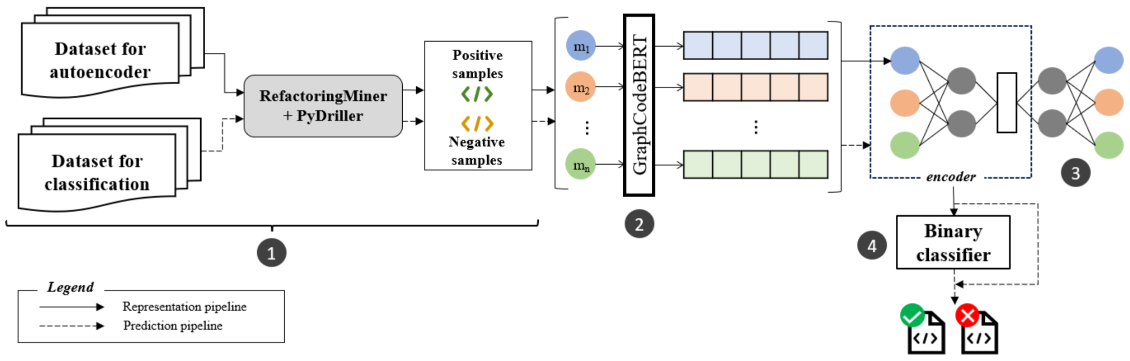


Figure 3.1: Overview of the proposed approach

a benchmark for automated refactoring candidate identification approaches properly. To study the effectiveness of the method representation generated using GraphCodeBERT in a binary classification task, we propose an Autoencoder-based approach to identify latent features.

Replication package: We made our code [76] publicly available with our training data [77] for easier replication and use.

3.2 Approach

This section describes the experimental approach followed to investigate the potential of Large Language Models (large language models) in determining the suitability of a method for *extract method* refactoring.

3.2.1 Overview

The study aims to develop a DL-based *extract method* refactoring candidate identification technique that addresses the deficiencies in the existing studies. Figure 4.1 presents an overview of our approach. We first pick a set of repositories to prepare our dataset. We use existing tools RefactoringMiner [109] and PyDriller [97], to segregate methods into positive and negative samples. Our approach utilizes GraphCodeBERT to generate embeddings for each sample. We employ a DL model based on Autoencoder [60] that is used for feature extraction and dimensionality reduction. We utilize the encoder component of the trained Autoencoder to generate a lower-dimensional latent space representation from the initially high-dimensional embedding input. The representation obtained from the bottleneck layer of Autoencoder is then used as a feature vector to train a *Random Forest* (RF) classifier on the *extract method* identification task. We formulate the following research questions.

RQ1 *How does our proposed approach perform compared to the state-of-the-art?*

By answering this research question, we intend to evaluate and validate the performance of the proposed approach as compared to the state of the art.

RQ2 *How effectively does the autoencoder extracts features for the classification task?*

In this research question, we aim to evaluate the effectiveness of the employed autoencoder-based model by extracting the learned features and using them for the classification task.

3.2.2 Dataset preparation

We utilized a subset of repositories (5%) from the 11,149 open-source Java repositories used by Aniche *et al.* [7], as working with the entire set required extensive computing infrastructure. This initial selection yielded 558 repositories, from which we excluded repositories that were no longer available on GitHub or lacked any instances of *extract method* refactoring throughout their history. After filtering, we obtained 410 repositories with at least one *extract method* refactoring performed. To leverage a trained autoencoder and using the trained encoder of the autoencoder for the classification task without data leakage, we divided this dataset into two parts: one for autoencoder pipeline with 208 repositories and the remaining 202 for the classification pipeline.

As shown in step ❶ of Figure 4.1, we use RefactoringMiner, a state-of-the-art refactoring detection tool [109, 107] to prepare our dataset. This tool reports performed refactorings, if any, in each commit within a Java repository's history. It provides essential metadata such as the code component involved (*e.g.,* method start line and end line in the case of *extract method* refactoring) and the associated commit hash. Leveraging this information, we utilize PyDriller [97] to iterate through the identified commits and extract source code of the involved methods.

We identify positive samples where *extract method* refactoring has been applied following the mechanism described above. Identifying negative samples for *extract method* refactoring is a challenging task. Merely excluding methods not reported by RefactoringMiner is insufficient since the absence of refactoring does not guarantee a method is not a refactoring candidate. Previous work have proposed heuristics-based approaches to address this challenge. For instance, Aniche *et al.* [7] used a criterion based on the method's modification history, while Yamanaka *et al.* [124] selected code portions that differ from actual extractions. However, these heuristics may introduce noise and create sub-optimal datasets by misclassifying potential *extract method* candidates.

We propose a new mechanism to identify negative samples for the study. A method is considered a negative sample in commit C_n if it underwent *extract method* refactoring in its parent commit C_{n-1}. The rationale behind this idea is that it is highly unlikely that a method that underwent *extract method* refactoring will again go through the same refactoring. This reduces the risk of false negative detection and ensures a high quality dataset. The aforementioned approach of identifying positive and negative samples, resulted in $27,634$ and $27,796$ samples for training and evaluating the Autoencoder, and binary classifier respectively. Table 3.1 describes the dataset statistics used to train, test and validate our approach.

Table 3.1: Dataset statistics

Dataset	Positive samples			Negative samples		
	Avg. LOC	Avg. token length	Median LOC	Avg. LOC	Avg. token length	Median LOC
Train split	36.46	343.26	14	26	25.23	263.24
Test split	36.22	341.57	14	26	24.46	261.55
Val split	35.57	338.88	13	26	25.50	268.27

3.2.3 Data representation

In step ❷, we use GraphCodeBERT to capture both syntactic and semantic information of code, providing a comprehensive representation of code snippets by using graph-guided masked attention function to incorporate the code structure. The initial step in processing the input code through the GraphCodeBERT model involves tokenization and encoding. To accomplish this, we utilize the pre-trained GraphCodeBERT tokenizer. To ensure the token sequence adheres to the model's maximum length of 512, we truncate it if it surpasses this limit. Subsequently, we perform batch encoding on the token sequence, generating `input_ids` which represent the tokens numerically for the model.

To extract the embeddings, we pass this encoded input to GraphCodeBERT. During the forward propagation of the input, each of the 12 hidden layers of the model generates individual token embeddings based on the surrounding context. To get the condensed representation of the sequence of tokens, we use mean pooling. We conducted a pilot study and we found that this approach performs better than taking the embedding of the `[SEP]` token alone. This results in a single embedding vector of size 768 for each of the input sample. We consider this as our feature vector for the classification task.

3.2.4 Model training and classification

Autoencoder

We use the generated embeddings from GraphCodeBERT as the input for training the autoencoder model (step ❸). The architecture of the autoencoder that we trained consists of an encoder with three fully connected linear layers and ReLU activation to learn the hidden representation that reduces the input dimension to a bottleneck layer of size 128. The decoder reverses this process to reconstruct the original input of size 768. The autoencoder model is trained on 70%. The rest 30% is used to validate the model. We calculate the reconstruction loss using *Mean Squared Error* (MSE) loss.

Binary classifier

After training the autoencoder, in step ❹, we take the *encoder* part of the trained model and use it as our feature extractor for the binary classification dataset. We train two classifiers—a traditional machine learning model *Random Forest* and a DL-based feed forward neural network, and compare their performance. We chose to use *Random Forest* due to its ensemble learning method for classification and its ability to learn the non-linear relationship between the features, *Random Forest* has shown to perform very well in different software engineering tasks [20, 19] including refactoring identification [7, 112].

To train our models, we first split the $27,796$ samples into train, validation, and test sets in $70 : 10 : 20$ ratio using stratified sampling. We use *GridSearchCV* to select the optimal hyper-parameters for *Random Forest*. The optimal set of hyper-parameter values along with their search space is reported in Table 3.2 The neural network classifier consists of two fully connected layers with ReLU activation and a final sigmoid activation layer.

Table 3.2: Optimal hyper-parameter values for random forest

Parameter	**Search space**	**Best value**
Number of trees	$[100, 200, 300, 1000]$	1000
Minimum samples split	$[8, 10, 12]$	10
Minimum leaf node samples	$[3, 4, 5]$	3
Maximum features	$[2, 3]$	2
Maximum tree depth	$[80, 90, 100, 110]$	80

Evaluation

To evaluate our models, we calculate the *accuracy, precision, recall,* and *F1 score.*

Initially, the test split of our classification dataset, contains positive and negative samples in equal proportion. However, it has been argued [92, 20] that a test set not representative of the real-world may show good performance while experimentation but do poorly when deployed in a real-world scenario. To address this issue, we identify the ratio of positive and negative samples in the following manner. First, we sample 20 repositories from our dataset randomly. For each of the selected repositories, we identify the commits in which *extract method* refactoring has been applied using RefactoringMiner along with the count of such methods (*posCount*). Using PyDriller, we identify the count of total methods present in the source code for that commit (*totalCount*). Then we compute the ratio $\frac{posCount}{totalCount}$ and take the mean across all identified commits to find a real-world ratio of *extract method* refactoring candidates. We modify the test set to represent the computed ratio (85 : 15) and then perform the evaluation.

3.3 Experimental Results

RQ1: *How does our proposed approach perform compared to the state-of-the-art?*

In this research question, we compare our two approaches `M1` (*i.e.,* neural network-based classification) and `M2` (*i.e., Random Forest*-based classification), where both the models utilize GraphCodeBERT and Autoencoder to generate latent representation. We compare the results from our models against state-of-the-art approach from Aniche *et al.* [7]. Though the baseline study compare many machine learning techniques, we chose to compare our models with only their *Random Forest* model because it reported the best results in that study. All of the models we tested using the same test split.

Table 3.3: Experimental results for RQ1

Models	**Accuracy**	**Precision**	**Recall**	**F1-score**
M1 (GraphCode-BERT + Autoencoder + Neural Network)	0.57	0.71	0.57	0.63
M2 (GraphCode-BERT + Autoencoder + Random forest)	**0.87**	**0.90**	**0.87**	**0.88**
Baseline (with random forest)	0.84	0.44	**0.87**	0.58

Table 4.3 presents results of our experiments. From the results it is evident that our `M2` model outperforms the baseline as well as the `M1` model. We observe that both `M1` and `M2` outperform the baseline. Specifically, we see that `M2` outperform the *Random Forest* used by Aniche *et al.* by nearly 50% in terms of precision. At the

same time, our model `M2` exhibits a good recall rate of 0.87. Consequently, we see that our model performs significantly better than the considered baseline model by approximately 30% in terms of F1 score.

RQ1 Summary: Our results show that our *Random Forest*-based model outperforms the baseline model significantly (by 30%, in terms of F1 score). The results indicate that our code representation is successfully capturing syntactic and semantic characteristics of code necessary to identify *extract method* refactoring candidates.

RQ2: *How effectively does the autoencoder extracts features for the classification task?*

We train an autoencoder model and use the trained encoder part of it as a feature extractor. We do so to reduce the vectors' dimensionality and extract relevant features from the embeddings generated from GraphCodeBERT.

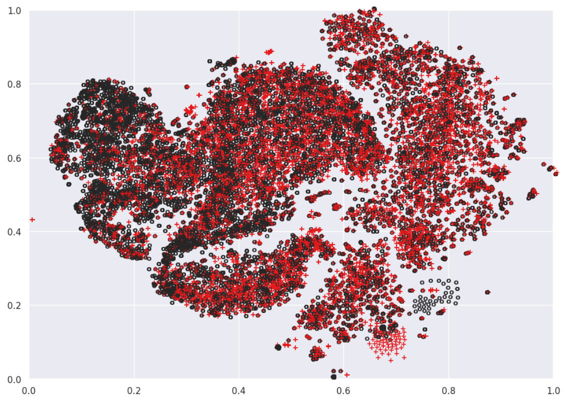


Figure 3.2: Class separation with embeddings (from GraphCodeBERT)

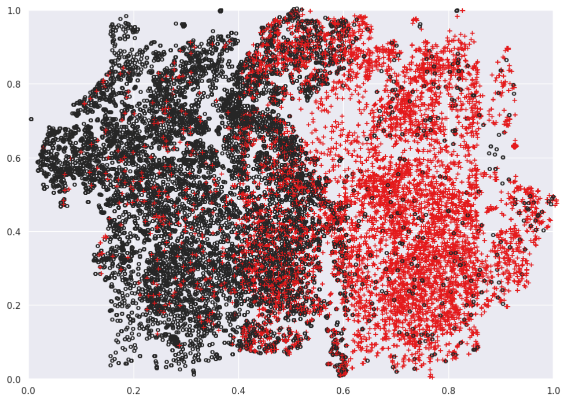


Figure 3.3: Class separation with encoded embeddings (from Autoencoder)

To measure the performance and usefulness of the trained autoencoder model as a feature extractor, we first analyze the `t-SNE` [114] plots of the embeddings generated by GraphCodeBERT and those generated by the combination of GraphCodeBERT and Autoencoder. We do so to study the class separability. The distinguishability of classes in the `t-SNE` space is assessed through a clear separation and quantified by calculating the Euclidean distance between the centroids of each class. Figure 3.3 shows a reasonably clear bifurcation between the classes with an euclidean distance of 0.367 as compared to 0.122 for Figure 3.2. We can infer that a classification model trained on the encoded embedding will perform better than the one trained on embeddings alone. This observation can be attributed to the autoencoder being able to learn hidden features unique to each type of sample which helps it to segregate them.

To further investigate the effectiveness of these representations, we perform an ablation study where we train classifier with and without the autoencoder representation of the embeddings. We report the performance results in Table 3.4. The *Random Forest* classification model, when trained with the encoded representation of the embedding provided by Autoencoder, outperforms the one trained with only the GraphCodeBERT embedding vector as features. These findings support our claim that the autoencoder extracts features and reduces dimensionality effectively in the context of refactoring candidate identification.

RQ2 Summary: Autoencoder-encoded code representations from GraphCodeBERT significantly improve classification performance compared to GraphCodeBERT representations alone.

Table 3.4: Encoded embedding performance

Models	**Accuracy**	**Precision**	**Recall**	**F1-score**
Embeddings	0.57	0.71	0.57	0.63
Encoded embeddings	**0.87**	**0.90**	**0.87**	**0.88**

3.4 Threats to Validity

Construct validity: Construct validity measures the degree to which tools and metrics actually measure the properties that they are supposed to measure. It concerns the appropriateness of observations and inferences based on measurements taken during the study. The quality of positive and negative samples extracted from RefactoringMiner and PyDriller poses a threat to validity. Though RefactoringMiner and PyDriller are state-of-the-art tools that are widely used and considered accurate, to mitigate the threat, we randomly picked up a subset of the identified negative samples and manually evaluated the quality of the samples.

Our approach uses a significantly smaller dataset (approximately 5%) compared to the dataset used in the state-of-the-art approach. Though the chosen repositories are representative, it can be considered a threat to validity. We chose the smaller dataset because of the computing resources required for the full-size dataset. Additionally, in this study, we aimed to explore the feasibility and effectiveness of the proposed approach. In the future, we aim to repeat the experiment with a larger dataset.

External validity: External validity concerns the generalizability and repeatability of the produced results. One of the threats to validity in this thesis is that the approach proposed is exclusive to *extract method* refactoring. Using our approach for another kind of refactoring is challenging and requires extensive reworking of the approach used. For example, *move method* refactoring moves a method to an appropriate class [27]. We cannot apply the same approach that we adopted to create *extract method* dataset for *move method* refactoring because we will not have the code to collect in the refactored commit since the method would have moved to another class. A similar challenge is expected when considering other refactoring types, such as *pull up method* and *push down method.* In the future, we would like to address this challenge and propose an effective and generic dataset-creation approach for different refactoring types.

Chapter 4

Comparing Reinforcement Learning and Supervised Learning in generating refactored code accurately

Learning without thought is labour lost;
thought without learning is perilous.
— Confucius

In this chapter, we present an innovative approach that combines reinforcement learning with large language models to fully automate the extract method refactoring process. Our method addresses key limitations of current tools by considering code-specific requirements during the refactoring process. The results show notable improvements in both automated metrics and practical code functionality, demonstrating the potential for reducing manual developer effort in code maintenance.

4.1 Introduction

Refactoring is an important software development activity that employs various techniques to enhance the structure and quality of source code without altering its functionality [75, 27]. By removing code smells [27] the practice aims to improve maintainability, encapsulating quality attributes such as readability, flexibility, and testability [22, 13]. Refactoring helps maintain high code quality, facilitating long-term maintainability and evolution [69].

Extract method refactoring is one of the most commonly applied refactoring techniques that involves moving a coherent code fragment from a method into a new, aptly named method [27]. By creating cohesive and smaller methods, extract method refactoring not only improves code quality and maintainability but also serves as a foundation for more complex refactoring operations [129]. Extract method refactoring constitutes a significant proportion, approximately 49.6%, of the total refactoring recommendations generated by JDeodorant [104], a widely recognized tool for supporting extract method operations. Furthermore, this refactoring technique has been acknowledged as a crucial operation by both open-source developers [96] and industry practitioners [113], underscoring its importance in software maintenance.

Automatically performing extract method refactoring, consist of two major steps [50]. First, *identification* of a candidate method that requires extract method refactoring; and second, intelligently *extracting* the logic and *forming* a new method with appropriate parameters, without human intervention. For the first step, *i.e.,* identifying a candidate method for the refactoring, practitioners often rely on intuition and experience. They also utilize automated tools to assess code quality metrics and detect code smells [94] to get aid in the decision process. The second step of automated extract method refactoring involves comprehending and extracting source code into a new method. Several approaches have been proposed to address this challenge. Hubert [38] developed a method for generating extract method refactoring candidates using static code analysis tools. Maruyama [38] proposed a candidate generation technique utilizing block-based slicing. Shahidi *et al.* [90] introduced an algorithm for identifying, generating, and ranking extract method candidates through graph analysis. However, these approaches exhibit a few limitations. Specifically, most of these approaches require the developers to manually identify the bounds of a block to be refactored *i.e.,* start and end statements, to perform the refactoring. Such a reliance on human knowledge reduces the efficacy and significance of automated refactoring. Furthermore, static analysis and metric-based methods often fail to capture latent contextual and syntactical code characteristics that could enhance the refactored code. For instance, these approaches do not offer meaningful identifiers for the new method and its parameters.

The emergence of large language models (LLMs) has enabled the convenience in generative tasks, including code generation with high accuracy [121, 2]. The field of code generation has seen significant advancements recently, with pre-trained language models such as GitHub Copilot [31] and Amazon Q Developer [8] demonstrating impressive capabilities. Though such LLM perform well on many text and code generation tasks, they show mediocre performance for tasks requiring domain-specific or uncommon knowledge. For example, LLM have shown proficiency in generating code for known and common problems but they struggle with unfamiliar problems [15]. Similarly, in our context, current LLM can generate refactored code but often omit the contextual information or generate incomplete, broken, or even uncompilable code [40]. Moreover, using third-party code completion services raises privacy concerns for many organizations. A notable example is Samsung Electronics [39], which reportedly experienced three data leakage incidents while using online code completion tools such as ChatGPT. These issues highlight the growing need for developing task-specific code generation models.

Language models for code are sequence-to-sequence models pre-trained on large corpus of code and can be fine-tuned for various software engineering tasks, including code summarization [3, 99], translation [24, 126], completion [9, 23], bug localization [125], vulnerability detection [62, 130], and program repair [93]. Despite these advancements, to the best of our knowledge, the application of language models for code refactoring remains largely unexplored. Inspired by applying LLMs on a variety of software engineering tasks, there has been some attempts to generate refactored code using them. For example, a recent contribution by Pomian *et al.* [80] introduced EM-Assist, an IntelliJ IDEA plugin that leverages LLMs to generate and rank refactoring suggestions using few-shot prompting.

Fine-tuning is another common technique to train a pre-trained model for a specific downstream task. While fine-tuning pre-trained code language models appears to be a promising solution, it has been observed that a considerable portion of programs generated by these models often fail to pass unit tests [15, 58, 40]. Such challenges deter the adoption of the automated refactoring tools and methods.

To address these challenges, we evaluate performance of fine-tuned models and propose a deep reinforcement learning approach that aligns fine-tuned code language models to generate refactored code by applying extract method refactoring automatically. Our approach, first, creates a dataset using state-of-the-art tools such as RefactoringMiner [109, 107]. We use the dataset to fine-tune four language models, pre-trained on code, using Supervised Fine Tuning (SFT) [37, 46] To enhance model performance and better align it with the objective of generating compilable code while preserving functionality, we use Proximal Policy Optimization (PPO) [86] for reinforcement learning optimization.

Our reinforcement learning approach utilizes an actor-critic architecture [49, 116], where the actor component generates refactored code, and the critic component assesses the quality of the generated code. This architecture enables the model to learn more efficiently in the complex space of code refactoring by providing guidance on the desirability of different refactoring decisions. The critic component incorporates discrete, non-differentiable reward signals in three stages. We first check for syntactic correctness, then assess whether the code compiles successfully, and finally, we use RefactoringMiner to detect if the desired refactoring has been applied.

To strike a balance between generating refactorings and maintaining the knowledge gained during supervised fine-tuning, we introduce a Kullback-Leibler (KL) divergence [51, 95] term in the reward function. This term measures the difference between the model's current behavior and its initial behavior learned during supervised fine-tuning. By incorporating this term, we encourage the model to explore new refactoring strategies while preventing it from deviating too far from its initial understanding of code refactoring.

Our study yielded promising results. The PLBART model, when fine-tuned using supervised learning, demonstrates superior performance among the chosen models when evaluated using conventional metrics such as BLEU, ROUGE, and CodeBLEU. However, Code-T5 outperforms other models when trained with deep reinforcement learning. **We observe that combining supervised fine-tuning with deep reinforcement learning prove most effective**, compared to fine-tuning the models or training using reinforcement learning individually. Qualitative evaluation further validates that the combination works the best, exhibiting enhanced syntactic accuracy, compilation rates, and unit test performance.

We list the key contributions of this thesis below.

- We evaluate the effectiveness of supervised fine-tuned models for automatic *extract method* refactoring. The approach addresses the limitations of existing approaches such as manual code selection to specify the code block to-be extracted.

- The study presents a hybrid method that combines supervised fine-tuning with reinforcement learning optimization, specifically tailored for extract method

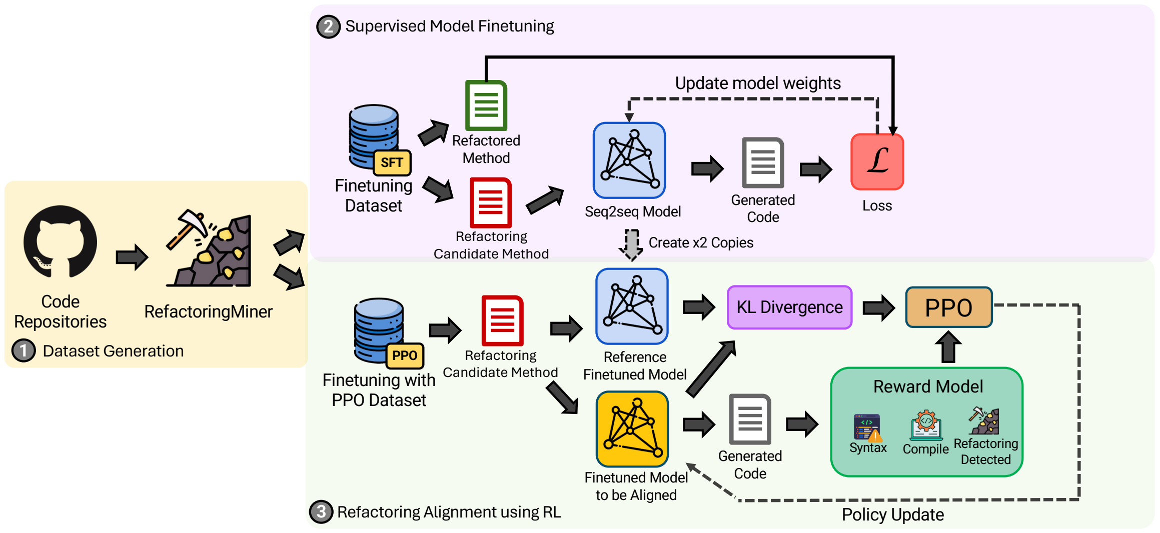


Figure 4.1: Overview of the proposed approach.

refactoring tasks. We then evaluate the approach both quantitatively and qualitatively to ensure that it generates syntactically and semantically accurate refactorings.

- This study also contributes a tool for analyzing Java repositories on GitHub to create an extract method refactoring datasets with associated metadata. We provide the tool and the dataset created using it for replication and extension purposes.

4.2 Methods

This section details the goal, research questions, and the approach, including setup, and metrics used to rigorously test and validate our proposed method.

4.2.1 Overview

The goal of this study is to evaluate the effectiveness of fine-tuned LLMs pretrained on code and develop a deep reinforcement learning-based approach for generating code for extract method refactoring. We seek to demonstrate the effectiveness of our approach not only quantitatively but also qualitatively. We formulate the following research questions.

RQ1. *How does supervised fine-tuning perform for extract method refactoring task?*

By answering this research question, we aim to evaluate how well does supervised fine-tuning a code large language model perform in automatically performing *extract method* refactoring.

RQ2. *How well does a reinforcement learning approach perform for automating extract method refactoring?*

This question examines whether code large language models can be directly aligned using reinforcement learning techniques to effectively perform *extract method* refactoring.

RQ3. *How does a reinforcement learning approach, combined with fine-tuned large language models, perform for automating extract method refactoring?*

This question assesses the impact of combining Proximal Policy Optimization (PPO) with custom reward signals on a fine-tuned model's performance in *extract method* refactoring tasks.

Figure 4.1 illustrates our methodology. We create our dataset by using the tools such as SEART tool [18] and RefactoringMiner [107, 109]. Following dataset preparation, we fine-tune four different models: Code-T5 and PLBART, which are encoder-decoder models, and CodeGPT-adapt and CodeGen, which are decoder-only models. We evaluate the performance of these models using both quantitative and qualitative measures. After conducting both quantitative and qualitative evaluations, we align the pre-trained model directly using the Proximal Policy Optimization (PPO) algorithm. Subsequently, we align the fine-tuned model using the same PPO algorithm. We systematically evaluate the applied approach using standard evaluation metrics. We also evaluate the models qualitatively using three key checks *i.e.,* syntactic validity, compilability, and the presence of the desired refactoring in the generated code.

4.2.2 Dataset Creation

We employ a systematic approach to identify and collect extract method refactoring instances across multiple open-source Java repositories. Step ❶ in Figure 4.1 shows an overview of the dataset preparation pipeline. We use SEART [18] tool to select a list of repositories for analysis. SEART tool is a GitHub project sampling tool, offering various commonly used filters (such as number of commits and stars). We obtain a list of all non-forked Java repositories created between 2013 and 2023, that are active in 2024, have at least 100 commits, and minimum 50 stars. We obtained a total of $1,618$ repositories satisfying the criteria.

Algorithm 1 Procedure for Creating Dataset

```
 1: Input: List of repositories $R = \{r_1, r_2, \ldots, r_{\cdot n}\}$
 2: Output: JSONL file with keys “Input” and “Output”
 3: procedure CreateDataset($R$)
 4:     $Data \leftarrow \emptyset$                    ▷ Initialize the dataset as an empty set
 5:     for each repository $r_i \in R$ do
 6:         Retrieve branch details for $r_i$
 7:         Fetch the list of commits for the given branch
 8:         for each commit $c_j$ in the list of commits do
 9:             Identify refactorings performed in $c_j$
10:             if extract method refactoring is detected then
11:                 Extract metadata associated with the refactoring
12:                 Extract the refactored method using the metadata
13:                 Checkout to the previous commit $c_{j-1}$
14:                 Extract the original method from $c_{j-1}$
15:                 Create output JSON object
16:                 Append this JSON object to $Data$
17:             end if
18:         end for
19:     end for
20:     Store $Data$ in a JSONL file
21: end procedure
```

To iteratively process the list of repositories to prepare the dataset, we created a custom Command Line Interface (CLI) tool. Algorithm 1 provides a pseudocode of the functionality of the tool. For each repository, we retrieve branch details and fetch the commit history. We then iterate through each commit, identifying any extract method refactorings performed using RefactoringMiner [109, 107]. When such a refactoring is detected, the algorithm extracts relevant metadata and the refactored method from the current commit c_j. It then checks out the previous commit, c_{j-1} to extract the original, pre-refactored method. This pair of pre- and post-refactoring methods, along with associated metadata (such as file path, class content and start and end line of the methods), is packaged into a JSON object. These JSON objects are accumulated into an array, which is ultimately stored in a JSONL file format. This approach enables the creation of a comprehensive dataset ($\mathcal{D}$) that captures the before and after states of extract method refactorings across multiple repositories.

Table 4.1: Dataset statistics

Dataset	Before pre-processing		After pre-processing	
	Avg. source token length	Avg. target token length	Avg. source token length	Avg. target token length
$\mathcal{D}_{SFT}$	412.77	446.13	184.26	241.63
$\mathcal{D}_{RL}$	410.60	449.09	187.62	242.13

For RQ1 and RQ2, we use the entire dataset. For RQ3, we divide the dataset into two mutually exclusive subsets one for supervised fine tuning and the other for the aligning the fine-tuned model with deep reinforcement learning. We divide the dataset to maintain data integrity and avoid data leak while training for RQ3. We divide the repository list of $1,618$ repositories, collected from the SEART tool, in half. We applied the aforementioned procedure to process both sets of repositories. This resulted in $38,441$ samples for the supervised fine tuning ($\mathcal{D}_{SFT}$) and $9,313$ samples for deep reinforcement learning ($\mathcal{D}_{RL}$). However, the resulting datasets contained samples that exceeded the context window (maximum input sequence length) of our selected fine-tuning models. Among these models, `Code-T5` has the smallest context window of 512 tokens, while others support up to $2,048$ tokens. To ensure compatibility across all models, we use 512 as our maximum context length, eliminating any samples that surpassed this 512-token threshold. After pre processing, $\mathcal{D}_{SFT}$ contains $26,949$ samples and $6,528$ samples in $\mathcal{D}_{RL}$. Table 4.1 presents the average token length distribution for both the datasets. Finally, each of the dataset is divided in 70 : 20 : 10 ratio for training, testing and validation.

4.2.3 Training Models

Fine tuning LLMs

We employ the following criteria to select the models for fine-tuning. The selected models must belong to encoder-decoder or decoder-only architecture. We exclude encoder-only models, such as CodeBERT, from our study because the encoder-only models are not well-suited for sequence-to-sequence (seq2seq) generation tasks [120]. Encoder-only model architectures like BERT are designed to understand input sequences but lack the ability to generate new ones. They're optimized for tasks like classification or feature extraction, not for producing variable-length outputs required in seq2seq tasks. Without a decoder component and autoregressive generation capability, these models can't effectively perform tasks such as translation or text generation that require producing new sequences based on input. We select the following models, two belonging to encoder-decoder and two to decoder-only architecture family, based on the the above-mentioned criteria.

- **Code-T5**: Code-T5 [121] is a pre-trained encoder-decoder model that incorporates token type information from code and employs an identifier-aware pre-training objective to better utilize identifiers. Code-T5 offers a unified framework that supports both code understanding and generation tasks, enabling multi-task learning. This model has been successfully applied to various code related tasks such as code summarization [5, 32], code translation [54] and vulnerability detection [79, 36].

- **PLBART**: PLBART [2] is a pre-trained sequence-to-sequence model that can perform a wide range of program and language understanding and generation tasks. It is trained on a large dataset of Java and Python functions along with their associated natural language text using denoising autoencoding. PLBART has been used in various software engineering applications especially in program repair task [79, 122]

- **CodeGPT-adapt**: CodeGPT-adapt [63] is a GPT-2-based decoder-only Transformer model for code completion, pre-trained on Python and Java code from CodeSearchNet datasets. It learns code structure and syntax through pre-training, enabling it to generate code automatically. It has been widely used for code generation tasks such as code completion [57, 36].

- **CodeGen**: CodeGen [74] is a Transformer-based autoregressive language model trained on natural language and programming language datasets. It employs next-token prediction as its learning objective and has shown outstanding performance in program synthesis tasks [16].

Step ❷ in Figure 4.1 illustrates the SFT strategy employed for extract method refactoring. To train the encoder-decoder models, the pre-refactored code is first tokenized to serve as the input sequence. After a forward pass through the model, output tokens are generated and decoded using the same tokenizer. The resulting method is then compared to the ground truth, which includes both the extracted method and the modified original method post-refactoring. The model weights are updated based on the cross-entropy loss computed between the predicted and ground truth methods.

For decoder-only models, the training process is similar, with the key difference being in the format of the input. In this case, the input sequence is formed by concatenating the pre-refactored code and the ground truth output, separated by a special [SEP] token. This format enables the model to learn from both the context of the original code and the desired output sequence in a single input representation.

Aligning the models with RL

In this study, we fine-tune and align the selected large language models for *extract method* refactoring using RL techniques (step ❸). We model the code transformation problem as a Markov Decision Process (MDP). We define the *state* as the set of all possible code representations and the state transition function as appending the chosen refactored token to the current sequence.

Algorithm 2 describes the pseudocode for aligning the fine tuned language model for extract method refactoring task. The algorithm starts with an initial policy (decision-making strategy) and a value function (which estimates how good a particular state is). It then goes through multiple training iterations to improve these over time. In each iteration, we sample a batch of code snippets from our RL dataset. For each snippet, *i.e.* the pre-refactored method, we use the current policy to generate a sequence of refactoring actions. To assess the quality of the sequence generated at each training step, we compute a reward based on three factors: syntactic correctness, compilation success, and whether the action is recognized as a valid refactoring.

The reward function plays a crucial role in evaluating the quality and correctness of the refactoring suggestions produced by the model. Our reward function consists of three key components, each addressing a specific aspect of the refactoring process:

1. *Syntactic Correctness:* We assess the presence of errors in the refactored code. For this purpose, we check the presence of error nodes in the Abstract Syntax Tree (AST) generated by *tree-sitter* of the generated code.

$$R_{syntax} = \begin{cases} +1 & \text{if no error nodes} \\ -1 & \text{if error nodes present} \end{cases} \tag{4.1}$$

2. *Compilation Success:* We verify whether the refactored code compiles successfully. While the compiler automatically checks for syntactic issues, separating syntactic correctness from compilation success allows us to provide the RL model with more granular feedback. This distinction is important because refactored code might be syntactically correct but still fail to compile due to semantic errors.

$$R_{compile} = \begin{cases} +1 & \text{if code compiles} \\ 0 & \text{if code fails to compile} \end{cases} \tag{4.2}$$

3. *Refactoring Detection:* We validate the presence of extract method refactoring in the generated code using RefactoringMiner.

$$R_{detect} = \begin{cases} +1 & \text{if detected by RefactoringMiner} \\ -1 & \text{if not detected} \end{cases} \tag{4.3}$$

The sum of these individual components gives us the total reward for a given refactoring suggestion.

$$R_{total} = R_{syntax} + R_{compile} + R_{detect} \tag{4.4}$$

This reward function encourages the language model to generate syntactically correct, compilable code that successfully implements the extract method refactoring.

The value head is used to estimate the value of the current state using the value function as shown in Equation 2.7. The algorithm then calculates how much better or worse each action was than expected (the *advantage*). This information is used to update the policy, aiming to increase the probability of actions that led to high rewards. However, to ensure stable learning, the algorithm checks how much the new policy differs from the old one using a measure called KL divergence as described in equation 2.8. If the difference is too large, the update is adjusted to prevent drastic changes. Finally, the value function is updated to better predict future rewards. By repeating this process many times, the algorithm gradually improves its ability to make good refactoring decisions.

Fine tuning setup.

We fine-tune the supervised model for 10 epochs on the dataset $\mathcal{D}$ for *RQ1*, and on the dataset $\mathcal{D}_{SFT}$ for *RQ3*. The training is conducted with a global batch size of 16, using the Adam optimizer [48] with an initial learning rate of 1.33×10^{-5}. For aligning the models with RL, we utilize and extend the TRL Python library, which is widely used for training transformer language models with reinforcement learning. The generation parameters are set with *min_tokens* as -1 and *max_tokens* as 512. The training consists of $20,000$ steps, with the model undergoing 10 PPO optimization epochs for each step. The code processes one batch per training step, generating responses, calculating rewards, and optimizing the model using PPO. For each batch, multiple internal PPO optimization passes are performed before moving to the next batch. This process repeats for $20,000$ steps, cycling through the dataset if necessary. A global batch size of 16 is maintained, and the Adam optimizer [48] is employed. We apply Adaptive KL control with an initial KL coefficient of 0.2. All experiments are conducted with a fixed seed value to ensure reproducibility and are performed on nodes of a High Performance Computing (HPC) cluster, utilizing 2 V100-32GB GPUs and 32 GB of RAM.

Algorithm 2 DRL Training for Extract Method Refactoring with KL Divergence

```
Require: Initial policy $\pi_\theta$, value function $V_\phi$, KL divergence coefficient $\beta$, weights
    $w_1, w_2, w_3$
 1: for each training iteration do
 2:     Sample batch of code snippets from dataset
 3:     for each code snippet $x$ do
 4:         Generate a refactored code snippet using current policy $\pi_\theta$
 5:         Compute syntactic correctness: $R_{syntax}$ using Eq. 4.1
 6:         Compute compilation success: $R_{compile}$ using Eq. 4.2
 7:         Compute refactoring validity: $R_{detect}$ using Eq. 4.3
 8:         Compute total reward: $R_{total} = w_1 \cdot R_{syntax} + w_2 \cdot R_{compile} + w_3 \cdot R_{detect}$
 9:         Estimate the value of the current state: $V_\phi(s)$
10:         Calculate advantage: $A = R_{total} - V_\phi(s)$
11:     end for
12:     Compute policy update to maximize:
13:         $J(\theta) = \mathbb{E}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A\right] - \beta \cdot \mathrm{KL}(\pi_{\theta_{old}} || \pi_\theta)$
14:     Apply the update to the policy: $\theta_{old} \leftarrow \theta$
15:     Update value function to minimize: $L(\phi) = \sum (R_{total} - V_\phi(s))^2$
16: end for
```

4.2.4 Evaluation

In this section, we summarize the metrics commonly used for code generation tasks. Also, we provide details about qualitative evaluation that goes beyond the standard metrics.

Evaluation Metrics

To assess the effectiveness of our models quantitatively, we utilize established metrics from natural language processing field BLEU [78] and ROUGE [59], as well as specialized metrics tailored for code evaluation, CodeBLEU [84] and syntax match score [132]. The widespread adoption of these metrics in academic research for evaluating generative models supports our decision to use them.

Qualitative Evaluation

To evaluate the effectiveness of our fine-tuned code language models in performing *extract method* refactoring, we construct a diverse test suite encompassing various complexity levels to ensure a thorough evaluation of the model's refactoring capabilities. To create the test cases for evaluation, we first identified pre-refactored original methods from the test sets of each of the dataset as mentioned in Section 4.2.2. We then selected 150 methods at random from $4,001$ test split samples ($2,695$ for SFT and $1,306$ for RL). Among these methods, few were very trivial like one or two liners and we discarded such methods. Finally, we collected 122 such methods which underwent extract method refactoring across various repositories.

A significant challenge in creating unit test cases is the lack of corresponding unit tests for many methods in the selected repositories. To address this, we leveraged gpt-4o (version: gpt-4o-2024-05-13) API [1] to generate unit tests and corresponding data. This approach aligns with recent research demonstrating the promising results of using language models for test case generation [110, 72]. We specifically employed the `ChatTester` framework proposed by Yuan *et al.* [127], to generate unit tests for our samples. The framework utilized the class context of the smelly method, extracted as per Algorithm 1, to create relevant unit test cases. The authors manually validated

[1]https://platform.openai.com/docs/models/gpt-4o

these generated test cases to ensure their quality and relevance. All the qualitative samples and corresponding test cases can be found in our replication package.

This combination of qualitative testing and quantitative analysis provides a systematic and objective assessment of our model's performance in extract method refactoring tasks. The multi-faceted evaluation approach allows for a comprehensive understanding of the model's capabilities and limitations across various *extract method* refactoring scenarios.

4.3 Results

4.3.1 RQ1: Performance of supervised fine-tuning for refactoring

The research question aims to evaluate the performance of the fine-tuned models for the refactored code generation. The first part of Table 4.2 (*i.e.,* PT + SFT column) presents the results obtained by the considered models for the refactored code generation task. The results presented in the table demonstrate that the PLBART model outperforms other models metrics and code-specific evaluation measures. Specifically, PLBART achieves the highest scores on the BLEU and ROUGE metrics, which assess the lexical and semantic similarity of the generated text to the ground truth. Crucially, PLBART also exhibits superior performance on the CodeBLEU metric, which captures the syntactic and structural fidelity of the generated code snippets.

Table 4.2: Experimental results for different learning objectives. Here, PT, SFT, and RL refer to pre-trained, supervised fine-tuned, and reinforcement learning-based models

Models	PT + SFT (RQ1)			PT + RL (RQ2)			PT + SFT + RL (RQ3)		
	BLEU	ROUGE	CodeBLEU	BLEU	ROUGE	CodeBLEU	BLEU	ROUGE	CodeBLEU
Code-T5	67.80	77.49	53.13	38.80	37.62	31.99	75.91★	79.92★	61.87★
PLBART	**68.28**	**80.62**	**55.66**	30.21	29.56	22.48	71.20	69.68	58.17
CodeGPT-adapt	62.68	65.76	49.29	27.68	30.76	20.29	64.96	67.82	47.99
CodeGen	59.32	63.74	42.11	34.32	33.74	27.11	61.59	60.52	46.67

Furthermore, a comparative analysis reveals that PLBART substantially outperforms the Code-T5 model, achieving a 4.54% higher CodeBLEU score. The performance gap is even more pronounced when contrasted with the CodeGPT-adapt model, for which PLBART demonstrates an 12.98% improvement on the CodeBLEU metric. These findings suggest that the PLBART model was successful in generating extract method refactored outputs that closely resemble the ground truth in terms of syntactic correctness.

To gain a more thorough understanding of the validity and robustness of the generated refactored outputs, additional validation checks and analyses are necessary. As detailed in Section 4.2.4, we create a manually validated dataset for qualitative evaluation. The dataset contains 122 samples with before and after refactored code and corresponding test cases. We use this qualitative dataset to check whether the trained model generates code without any syntactic and compilation errors, whether the generated code has extract method refactoring, and to what extent the generated code is passing the test cases. Table 4.3 presents the obtained results. For RQ1, notably, fine-tuned Code-T5 achieved the highest performance in qualitative evaluation. This supports our assertion that relying solely on quantitative metrics may yield misleading results and potentially produce low-quality refactored code.

RQ1 Summary: Fine tuning code large language models show an effective way to teach language models to generate refactored code automatically. Specifically, PLBART outperform other models in all the considered metrics. It show significant improvements over Code-T5 and CodeGPT-adapt, particularly in CodeBLEU scores. However, qualitative evaluation reveals that Code-T5 performs best in generating syntactically correct and functionally valid refactored code.

4.3.2 RQ2: Effectiveness of reinforcement learning for automated refactoring

This research question aims to evaluate the application of RL on the refactoring task when applied on pre-trained LLM. The second part of the Table 4.2 (*i.e.*, PT + RL column) shows the obtained results. Our results show that Code-T5 model demonstrates superior performance across all evaluation metrics compared to other language models when trained using RL.

Interestingly, unlike the results observed in RQ1 with traditional fine-tuning, direct fine-tuning using RL with Proximal Policy Optimization (PPO) does not perform well. This outcome may be attributed to the complexity of the *extract method* refactoring task and the potential mismatch between the RL objective and the nuanced requirements of code refactoring. Fine-tuning language models that have been pre-trained on tasks other than code refactoring directly using non-differentiable rewards poses challenges. The disparity between the pre-training task and the target task of code refactoring makes it difficult to effectively train the models using RL techniques.

Table 4.3: Qualitative evaluation of fine tuned models

	Models	**Syntactically correct (%)**	**Refactoring detected (%)**	**Compile successfully (%)**	**# of unit tests passed (out of 122)**
RQ1	Code-T5 (FT)	**78.6**	**66.4**	**72.1**	**41**
	PLBART (FT)	76.9	63.8	69.5	38
	CodeGPT-adapt (FT)	77.5	64.3	70.1	39
	CodeGen (FT)	78.3	65.1	71.2	40
RQ2	Code-T5 + RL	21.4	20.2	22.3	21
	PLBART + RL	21.7	18.9	21.5	16
	CodeGPT-adapt + RL	19.7	14.1	20.2	14
	CodeGen + RL	23.6	19.2	21.1	9
RQ3	Code-T5 (FT) + RL	**85.7**	**74.9**	**79.8**	**66**
	PLBART (FT) + RL	82.4	71.6	76.3	58
	CodeGPT-adapt (FT) + RL	83.1	72.2	77.5	61
	CodeGen (FT) + RL	84.3	73.5	78.6	63

Qualitatively also, as shown in *RQ2* of Table 4.3, the generated refactorings exhibit poor quality.

The RL method's poor performance in functional areas highlights a misalignment with the refactored code's true requirements. This suggests that the RL reward signals may insufficiently penalize syntactic and semantic errors, resulting in models to produce functionally valid code. A potential explanation for this behavior could be that the RL, performed on a generic pretrained language model, may not receive appropriate reward signals from our reward framework or the non-score rewards (KL divergence penalty). This hypothesis can be corroborated by examining Figure 4.2. Figure 4.2a illustrates the persistent high standard deviation of rewards for the RL fine-tuned model throughout increasing training steps. This trend indicates that the reward signals fail to effectively steer the model towards optimal performance. Concurrently, Figure 4.2b reveals an upward trajectory in the KL-Divergence penalty over time. This escalation suggests a growing divergence between the trained model and the reference model, further supporting our hypothesis that the current reward system may be inadequate for guiding the model towards generating functionally sound code refactorings.

RQ2 Summary: Generating refactored code from a pre-trained model directly aligned with RL does not produce comparable results to the corresponding fine-tuned models as shown in RQ1 quantitatively or qualitatively.

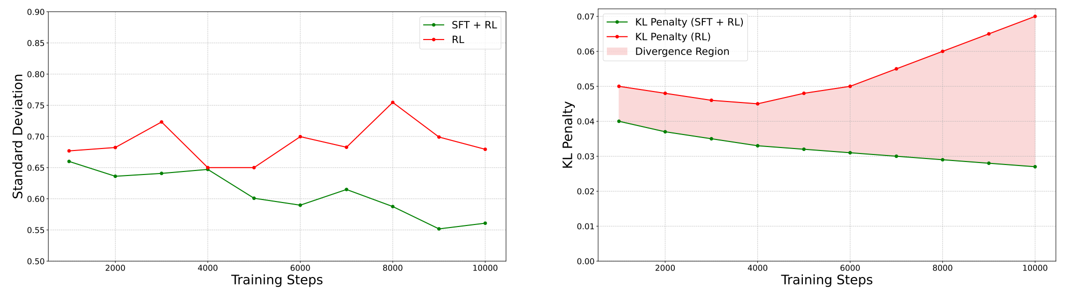
4.3.3 RQ3: Evaluating the performance of reinforcement learning and fine-tuned LLMs for refactoring

In this research question, we aim to evaluate the efficacy of applying RL to generate refactored code, focusing on model performance when fine-tuned using a combination of supervised fine-tuning (SFT) and RL objectives. Specifically, we start with the trained fine-tuned models from RQ1, and train them with PPO and reward from a feedback system to observe any improvements in the models compared to their fine-tuned counterparts.

Table 4.2 presents the results in the column titled PT + SFT + RL along with results obtained in other settings as discussed in RQ1 and RQ2. The results demonstrate that **the most effective outcomes are achieved when models are trained using both SFT and RL objectives.** This combined approach lead to significant improvements across various metrics. Specifically, we observed an approximate 10% increase in CodeBLEU compared to models trained solely with SFT, and an 11% improvement over those trained exclusively with RL. Similar performance gains were noted in other metrics, including BLEU and ROUGE. The superiority of the combined approach can be attributed to the complementary nature of SFT and RL. SFT excels at identifying inherent patterns and structures within data, primarily utilizing large labeled datasets. In contrast, RL adapts through environmental interactions, optimizing predefined reward metrics. By integrating these methodologies, our model can effectively navigate dynamic contexts while capturing underlying data patterns.

Our combined approach demonstrated a significant improvement in the evaluated quality metrics. The number of successfully passing test cases increased substantially, rising from 41 in the best-performing model, Code-T5, to 66—a significant improvement of approximately 61%. Additionally, RefactoringMiner identified an increased number of cases, from 87 to 98. These results highlight the efficacy of RL in producing accurate extract method refactored code.

Figure 4.2 illustrates the trends in the standard deviation of rewards and the KL-Divergence penalty across training steps. The initial decline in standard deviation, followed by stabilization, coupled with consistent KL-divergence penalties, suggests that our reward modeling strategy effectively aligns a fine-tuned language model for the extract method refactoring task. These results collectively highlight the efficacy of RL in producing accurate extract method refactored code.

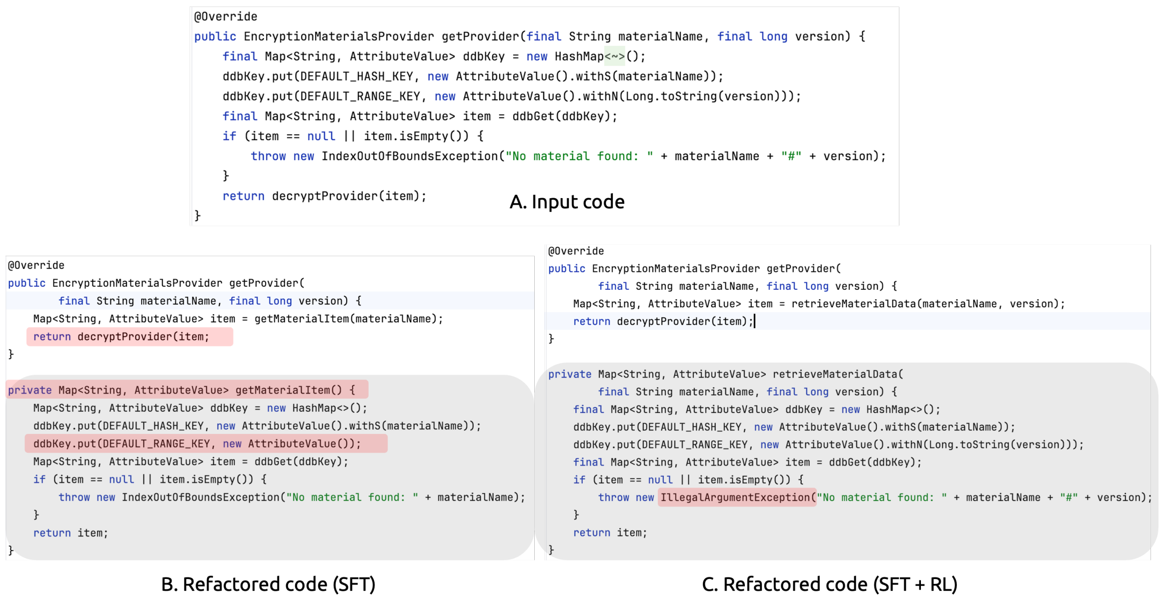


(a) Standard Deviation of Rewards (b) Negated KL Divergence Penalty

Figure 4.2: RL training observations

While SFT typically learns based on loss derived from labeled data, the integration with RL allows the model to benefit from a more comprehensive feedback system, including reward signals. This holistic approach contributes to the observed performance enhancements.

We present an illustrative example of a *extract method* refactoring in Figure 4.3, highlighting the differences of SFT and RL techniques. The original method belongs to `aws/aws-dynamodb-encryptionjava` repository as present in commit `ea43801`. Snippets `B` and `C` are generated by SFT model and combined SFT with RL aligned models respectively. As we can see from the generated example, there are few syntactic errors (highlighted by red background color) present in the output generated by the fine-tuned only model. The combined RL model seems to be more aligned to the ground truth. However, the generated code is not absolutely accurate because at line 10 it throws an `IllegalArgumentException` instead of `IndexOutOfBoundException`. But this example strengthens our claims that the combined SFT model with RL alignment enhances language model performance to generate more accurate extract method refactored code.



```java
@Override
public EncryptionMaterialsProvider getProvider(final String materialName, final long version) {
    final Map<String, AttributeValue> ddbKey = new HashMap<~>();
    ddbKey.put(DEFAULT_HASH_KEY, new AttributeValue().withS(materialName));
    ddbKey.put(DEFAULT_RANGE_KEY, new AttributeValue().withN(Long.toString(version)));
    final Map<String, AttributeValue> item = ddbGet(ddbKey);
    if (item == null || item.isEmpty()) {
        throw new IndexOutOfBoundsException("No material found: " + materialName + "#" + version);
    }
    return decryptProvider(item);
}
```

A. Input code

```java
@Override
public EncryptionMaterialsProvider getProvider(
        final String materialName, final long version) {
    Map<String, AttributeValue> item = getMaterialItem(materialName);
    return decryptProvider(item;
}

private Map<String, AttributeValue> getMaterialItem() {
    Map<String, AttributeValue> ddbKey = new HashMap<>();
    ddbKey.put(DEFAULT_HASH_KEY, new AttributeValue().withS(materialName));
    ddbKey.put(DEFAULT_RANGE_KEY, new AttributeValue());
    Map<String, AttributeValue> item = ddbGet(ddbKey);
    if (item == null || item.isEmpty()) {
        throw new IndexOutOfBoundsException("No material found: " + materialName);
    }
    return item;
}
```

B. Refactored code (SFT)

```java
@Override
public EncryptionMaterialsProvider getProvider(
        final String materialName, final long version) {
    Map<String, AttributeValue> item = retrieveMaterialData(materialName, version);
    return decryptProvider(item);
}

private Map<String, AttributeValue> retrieveMaterialData(
        final String materialName, final long version) {
    final Map<String, AttributeValue> ddbKey = new HashMap<>();
    ddbKey.put(DEFAULT_HASH_KEY, new AttributeValue().withS(materialName));
    ddbKey.put(DEFAULT_RANGE_KEY, new AttributeValue().withN(Long.toString(version)));
    final Map<String, AttributeValue> item = ddbGet(ddbKey);
    if (item == null || item.isEmpty()) {
        throw new IllegalArgumentException("No material found: " + materialName + "#" + version);
    }
    return item;
}
```

C. Refactored code (SFT + RL)

Figure 4.3: Extract method refactoring example generated using Supervised Fine-Tuning (SFT) and the combination of SFT and Reinforcement Learning (RL) techniques

RQ3 Summary: Our results demonstrates that combining supervised fine-tuning and RL objectives yields superior results in generating refactored code. This integrated approach outperforms individual methods, showing significant improvements in CodeBLEU, BLEU, and ROUGE metrics, while mitigating common limitations associated with single-objective training.

4.4 Discussions

While statistical metrics provide valuable insights, they may not fully capture a model's ability to generate high-quality code. We can observe the phenomenon in Table 4.2 and Table 4.3. The evaluation metrics used in Table 4.2 do not show very drastic difference in the RQ1 and RQ3 results. However, the qualitative results presented in Table 4.3 present very different narrative. We observe that the models trained using both supervised fine-tuning and RL techniques show significantly better results. Specifically, the number of test cases passed by the best model in RQ3 is 61% more than that of the best model in RQ1. This observation highlights the importance of qualitative evaluation in addition to traditional metrics-based evaluation.

One may wonder whether production-ready LLM for code, such as GitHub Copilot, can achieve better performance in automated extract method refactoring. Although this is beyond the scope of our study, it's reasonable to assume that such models could handle smaller, frequently performed refactorings. However, anecdotal evidence suggests these models may struggle with context-specific, complex, non-atomic refactorings. Further research is needed to examine the behavior of these models across various prompt settings and complexity levels.

4.5 Threats to Validity

Internal validity: Internal validity concerns relate to the reliability of conclusions drawn from our experimental results. To enhance the trustworthiness of our findings, we implemented several measures. Firstly, we addressed the potential confounding effect of varying hyperparameters by utilizing consistent settings across all models, based on the optimal configurations identified in prior research by Li *et al.* [57]. Additionally, we employed identical data splits for training and testing across all models, ensuring equal learning opportunities and evaluation conditions. These methodological decisions mitigate the risk of spurious results attributable to inconsistent experimental conditions, thereby strengthening the validity of our conclusions regarding the efficacy of deep reinforcement learning in generating refactored code methods.

External validity: External validity concerns in our study pertain to the generalizability of our findings beyond the Java context. Despite this focus, we argue that our methodology is highly transferable. Our data collection technique is language-agnostic, applicable to any refactoring scenario. The general-purpose models we employed, trained on vast code corpora, are adaptable to various programming languages. While these factors suggest broad applicability, further research across multiple languages and environments would be necessary to conclusively establish the universal validity of our approach.

Chapter 5

Conclusions & Future Work

The best way to predict the future is to invent it.

— Alan Kay

This thesis introduces a complete automated pipeline for extract method refactoring, encompassing both the identification of refactoring opportunities and their subsequent implementation. In this chapter, we summarize the results of our research, present the contributions of the thesis, elaborate our vision for future work, and conclude the thesis.

5.1 Summary of the Results

Automating software refactoring completely is an ambitious goal. To make meaningful progress in this direction, we focused our research on answering two fundamental questions. The first question addresses a common challenge developers face daily: identifying which code segments actually need refactoring. To tackle this challenge, we developed an innovative approach for identifying refactoring candidates. Our solution combines a self-supervised autoencoder with code representations from GraphCodeBERT, a pre-trained language model, to capture the essence of source code. We then trained a binary classifier to identify potential extract method refactoring opportunities. The results were promising - our approach outperformed existing machine learning techniques by 30% in F1 score, demonstrating its effectiveness in identifying refactoring candidates.

Chapter 4 of this thesis presents our comprehensive solution that incorporates reinforcement learning to automatically perform extract method refactoring. We began by building a substantial dataset of real-world extract method refactoring examples from Java projects. Our evaluation covered various large language models, including both encoder-decoder and decoder-only architectures. An interesting finding emerged during our analysis - while PLBART showed strong performance in traditional metrics like BLEU and ROUGE, its qualitative results fell short of expectations. This observation led us to develop an innovative hybrid approach that combines conventional fine-tuning with reinforcement learning alignment. This combination proved particularly effective, showing significant improvements across both quantitative and qualitative measures. The numbers tell a compelling story: our reinforcement learning-aligned approach improved upon traditional supervised fine-tuning by 11.96% in BLEU scores and 16.45% in CodeBLEU scores. Perhaps most importantly, we validated our approach through practical testing. When we ran our solution through a suite of 122 unit tests, the reinforcement learning-aligned Code-T5 model successfully passed 66 tests, compared to just 41 for the baseline model. These results clearly demonstrate our approach's ability to generate not just syntactically correct, but functionally valid refactorings.

5.2 Contributions of the Thesis

The main contributions of this research can be summarized as follows:

5.2.1 Research Contributions

- We solved a fundamental problem in the field by creating a framework that can reliably tell apart good and bad candidates for *extract method* refactoring. Our experimental results showed significant improvement in accuracy than what was previously possible, making it much more practical for real-world use.

- We developed a new way to understand code using Autoencoders with GraphCodeBERT. This new approach proved better at capturing the subtle non-heuristics based patterns in code that make refactoring decisions more accurate.

- By combining supervised learning with reinforcement learning, we created a hybrid system that's particularly good at automatically performing *extract method* refactoring. We observed significant performance improvement not only in terms of quantitative metrics but also validated our approach qualitatively using real world test cases.

5.2.2 Impact and Implications

Our work has some exciting practical implications for software development:

- Developers can now save significant time on refactoring tasks both while identifying what to refactor and how to refactor with reduced chance of human error.
- The high accuracy of our approaches and thorough validation of the generated results indicate that our solutions are ready for real-world use in development environments.
- Since we've made everything open source, the developer community can build upon our work, adapt it to their needs, and keep improving it.

These advancements mean that developers can spend less time on routine code maintenance and more time on creative problem-solving. The tools we've developed aren't just theoretical - they're practical solutions that can make a real difference in day-to-day software development work.

5.3 Future Work

5.3.1 Refactoring Candidate Identification

For refactoring candidate identification, our future work will focus on exploring advanced architectural variations of embedding models that can better capture code semantics and structural patterns. As an extension of our current work, we look forward to perform an ablation study to better understand the impact of each of the components and how modifying them can affect the outcome. We plan to investigate hierarchical attention mechanisms and hybrid approaches that combine traditional software metrics with learned representations. This includes developing more sophisticated methods for handling context-dependent refactoring opportunities and exploring multi-modal learning approaches that can leverage both source code and documentation.

5.3.2 Refactored Code Generation

In the domain of refactoring generation, we plan to enhance our reinforcement learning approach by developing more sophisticated reward functions that better capture code quality improvements and maintenance benefits. An in-depth analysis is required to better understand how the model performs in separating multiple concerns. We also plan to ensure the validity and reproducibility of our results by seeding and running the experiments multiple times. We aim to explore few-shot learning approaches that can adapt to project-specific coding standards and patterns, making the generated refactorings more contextually appropriate. Furthermore, we plan to make these techniques language and refactoring type agnostic.

5.3.3 Integrating these solutions to production

Looking ahead, our primary focus is to optimize the refactoring process for real-time performance, making it responsive enough to work alongside developers as they code. We aim to create a lightweight solution that maintains its high accuracy while integrating seamlessly with existing development environments. Beyond extract method refactoring, we plan to expand our approach to cover other important refactoring techniques. Our goal is to incorporate these capabilities into standard development workflows, making automated refactoring a natural and efficient part of the software development process.

This research lays the groundwork for more sophisticated automated code maintenance tools, opening new possibilities for improving software development practices.

Bibliography

[1] Mansi Agnihotri and Anuradha Chug. A systematic literature survey of software metrics, code smells and refactoring techniques. *Journal of Information Processing Systems*, 16(4):915–934, 2020.

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation, 2021.

[3] Toufique Ahmed and Premkumar Devanbu. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–5, 2022.

[4] Jehad Al Dallal. Identifying refactoring opportunities in object-oriented code: A systematic literature review. *Information and software Technology*, 58:231–249, 2015.

[5] Ali Al-Kaswan, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Premkumar Devanbu, and Arie van Deursen. Extending source code pre-trained language models to summarise decompiled binaries. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 260–271. IEEE, 2023.

[6] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019.

[7] Maurício Finavaro Aniche, Erick Galani Maziero, Rafael Serapilha Durelli, and Vinicius H. S. Durelli. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering*, 48:1432–1450, 2020.

[8] AWS. AI Coding Assistant - Amazon Q Developer - AWS — aws.amazon.com. https://aws.amazon.com/q/developer/. [Accessed 03-09-2024].

[9] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE):675–698, 2024.

[10] Osman Balci. Verification, validation, and accreditation. In *1998 winter simulation conference. proceedings (cat. no. 98ch36274)*, volume 1, pages 41–48. IEEE, 1998.

[11] Gabriele Bavota, Andrea De Lucia, and Rocco Oliveto. Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, 84(3):397–414, 2011.

[12] Sofia Charalampidou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Antonios Gkortzis, and Paris Avgeriou. Identifying extract method refactoring opportunities based on functional relevance. *IEEE Transactions on Software Engineering*, 43(10):954–974, 2016.

[13] Mandeep K Chawla and Indu Chhabra. Sqmma: Software quality model for maintainability analysis. In *Proceedings of the 8th Annual ACM India Conference*, pages 9–17, 2015.

[14] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling, 2021.

[15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[16] Fenia Christopoulou, Gerasimos Lampouras, Milan Gritta, Guchun Zhang, Yinpeng Guo, Zhongqi Li, Qi Zhang, Meng Xiao, Bo Shen, Lin Li, et al. Pangu-coder: Program synthesis with function-level language modeling. *arXiv preprint arXiv:2207.11280*, 2022.

[17] ISTVAN GERGELY Czibula and G Czibula. Hierarchical clustering based automatic refactorings detection. *WSEAS Transactions on Electronics*, 5(7):291–302, 2008.

[18] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for MSR studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pages 560–564. IEEE, 2021.

[19] S. Delphine Immaculate, M. Farida Begam, and M. Floramary. Software bug prediction using supervised machine learning algorithms. In *2019 International Conference on Data Science and Communication (IconDSC)*, pages 1–7, 2019.

[20] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. Detecting code smells using machine learning techniques: Are we there yet? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 612–621, 2018.

[21] Bart Du Bois, Serge Demeyer, and Jan Verelst. Refactoring-improving coupling and cohesion of existing code. In *11th working conference on reverse engineering*, pages 144–151, 2004.

[22] I Eee. Standard g lossary of softwareengineering terminology. *IEEE S o f t w are E n g ineerin g S tandards & oll ecti o n. I EEE*, pages 610–12, 1990.

[23] Aryaz Eghbali and Michael Pradel. De-hallucinator: Iterative grounding for llm-based code completion. *arXiv preprint arXiv:2401.01701*, 2024.

[24] Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, Joey Dodds, and Daniel Kroening. Towards translating real-world code with llms: A study of translating to rust. *arXiv preprint arXiv:2405.11514*, 2024.

[25] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[26] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, and Alexander Chatzigeorgiou. Identification and application of extract class refactorings in object-oriented systems. *Journal of Systems and Software*, 85(10):2241–2260, 2012.

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

[28] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

[29] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Identifying architectural bad smells. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 255–258. IEEE, 2009.

[30] Jan Gerling. Machine learning for software refactoring: a large-scale empirical study. Master's thesis, Delft University of Technology, 2020.

[31] GitHub. GitHub Copilot · Your AI pair programmer — github.com. https://github.com/features/copilot. [Accessed 03-09-2024].

[32] Jian Gu, Pasquale Salza, and Harald C Gall. Assemble foundation models for automatic code summarization. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 935–946. IEEE, 2022.

[33] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

[34] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang,

and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. *CoRR*, abs/2009.08366, 2020.

[35] Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International conference on learning representations*, 2019.

[36] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review, 2024.

[37] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*, 2018.

[38] Johannes Hubert. Implementation of an automatic extract method refactoring. Master's thesis, 2019.

[39] Contributing Writer Jai Vijayan. Samsung engineers feed sensitive data to chatgpt, sparking workplace ai warnings, Dec 2023.

[40] Akshita Jha and Chandan K Reddy. Codeattack: Code-based adversarial attacks for pre-trained programming language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 14892–14900, 2023.

[41] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International conference on machine learning*, pages 5110–5121. PMLR, 2020.

[42] Chitti Babu Karakati and Sethukarasi Thirumaaran. Software code refactoring based on deep neural network-based fitness function. *Concurrency and Computation: Practice and Experience*, 35(4):e7531.

[43] Anjan Karmakar and Romain Robbes. What do pre-trained code models know about code? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1332–1336, 2021.

[44] Yoshio Kataoka, Michael D Ernst, William G Griswold, and David Notkin. Automated support for program refactoring using invariants. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pages 736–743, 2001.

[45] Yaser Keneshloo, Tian Shi, Naren Ramakrishnan, and Chandan K. Reddy. Deep reinforcement learning for sequence to sequence models, 2019.

[46] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, volume 1, page 2, 2019.

[47] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[48] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[49] Vijay Konda and John Tsitsiklis. Actor-critic algorithms. *Advances in neural information processing systems*, 12, 1999.

[50] Mark Kramer and Philip H Newcomb. Legacy system modernization of the engineering operational sequencing system (eoss). In *Information Systems Transformation*, pages 249–281. Elsevier, 2010.

[51] Solomon Kullback. *Information theory and statistics.* Courier Corporation, 1997.

[52] Lov Kumar, Shashank Mouli Satapathy, and Lalita Bhanu Murthy. Method level refactoring prediction on five open source java projects using machine learning techniques. In *Proceedings of the 12th Innovations on Software Engineering Conference*, 2019.

[53] Zarina Kurbatova, Ivan Veselov, Yaroslav Golubev, and Timofey Bryksin. Recommendation of move method refactoring using path-based representation of code. *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020.

[54] Kusum Kusum, Abrar Ahmed, C Bhuvana, and V Vivek. Unsupervised translation of programming language-a survey paper. In *2022 4th International Conference on Advances in Computing, Communication Control and Networking (ICAC3N)*, pages 384–388. IEEE, 2022.

[55] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, 167:110610, 2020.

[56] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

[57] Bolun Li, Zhihong Sun, Tao Huang, Hongyu Zhang, Yao Wan, Ge Li, Zhi Jin, and Chen Lyu. Ircoco: Immediate rewards-guided deep reinforcement learning for code completion. *Proceedings of the ACM on Software Engineering*, 1(FSE):182–203, 2024.

[58] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[59] Chin-Yew Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

[60] Cheng-Yuan Liou, Wei-Chen Cheng, Jiun-Wei Liou, and Daw-Ran Liou. Autoencoder for words. *Neurocomputing*, 139:84–96, 2014.

[61] Hao Liu, Yanlin Wang, Zhao Wei, Yong Xu, Juhong Wang, Hui Li, and Rongrong Ji. Refbert: A two-stage pre-trained framework for automatic rename refactoring. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 740–752, 2023.

[62] Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software*, 212:112031, 2024.

[63] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021.

[64] Jonathan Mädler, Isabell Viedt, and Leon Urbas. Applying quality assurance concepts from software development to simulation model assessment in smart equipment. In *Computer Aided Chemical Engineering*, volume 50, pages 813–818. Elsevier, 2021.

[65] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[66] Katsuhisa Maruyama. Automated method-extraction refactoring by using block-based slicing. In *Proceedings of the 2001 symposium on Software reusability: putting software reuse in context*, pages 31–40, 2001.

[67] Davood Mazinanian, Nikolaos Tsantalis, Raphael Stein, and Zackary Valenta. Jdeodorant: clone refactoring. In *Proceedings of the 38th international conference on software engineering companion*, pages 613–616, 2016.

[68] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[69] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. A case study on the impact of refactoring on quality and productivity in an agile team. In *IFIP Central and East European Conference on Software Engineering Techniques*, pages 252–266. Springer, 2007.

[70] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2011.

[71] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[72] Noor Nashid, Mifta Sintaha, and Ali Mesbah. Retrieval-based prompt selection for code-related few-shot learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2450–2462. IEEE, 2023.

[73] Daniel Nichols, Pranav Polasam, Harshitha Menon, Aniruddha Marathe, Todd Gamblin, and Abhinav Bhatele. Performance-aligned llms for generating fast code. *arXiv preprint arXiv:2404.18864*, 2024.

[74] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[75] William F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[76] Indranil Palit, Gautam Shetty, Hera Arif, and Tushar Sharma. Extract method identification.

[77] Indranil Palit, Gautam Shetty, Hera Arif, and Tushar Sharma. Dataset for Automatic Refactoring Candidate Identification Leveraging Effective Code Representation, July 2023.

[78] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics.

[79] Rishov Paul, Md. Mohib Hossain, Mohammed Latif Siddiq, Masum Hasan, Anindya Iqbal, and Joanna C. S. Santos. Enhancing automated program repair through fine-tuning and prompt engineering, 2023.

[80] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Andrey Sokolov, Timofey Bryksin, and Danny Dig. Em-assist: Safe automated extractmethod refactoring with llms. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE '24. ACM, July 2024.

[81] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming.* John Wiley & Sons, 2014.

[82] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[83] Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. Sequence level training with recurrent neural networks, 2016.

[84] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis, 2020.

[85] Priyadarshni Suresh Sagar, Eman Abdulah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Christian D. Newman. Comparing commit messages and source code metrics for the prediction refactoring activities. *Algorithms*, 14(10), 2021.

[86] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[87] Robert C Seacord, Daniel Plakosh, and Grace A Lewis. *Modernizing legacy systems: software technologies, engineering processes, and business practices.* Addison-Wesley Professional, 2003.

[88] Carolyn Seaman and Yuepu Guo. Measuring and monitoring technical debt. In *Advances in Computers*, volume 82, pages 25–46. Elsevier, 2011.

[89] Gabriela Serban and Istvan-Gergely Czibula. Restructuring software systems using clustering. In *2007 22nd international symposium on computer and information sciences*, pages 1–6, 2007.

[90] Mahnoosh Shahidi, Mehrdad Ashtiani, and Morteza Zakeri-Nasrabadi. An automated extract method refactoring approach to correct the long method code smell. *Journal of Systems and Software*, 187:111221, 2022.

[91] Ashish Kumar Shakya, Gopinatha Pillai, and Sohom Chakrabarty. Reinforcement learning algorithms: A brief survey. *Expert Systems with Applications*, 231:120495, 2023.

[92] Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. Code smell detection by deep direct-learning and transfer-learning. *Journal of Systems and Software*, 176:110936, 2021.

[93] Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, Indira Vats, Hadi Moazen, and Federica Sarro. A survey on machine learning techniques applied to source code. *Journal of Systems and Software*, 209:111934, 2024.

[94] Tushar Sharma and Diomidis Spinellis. A survey on software smells. *Journal of Systems and Software*, 138:158 – 173, 2018.

[95] Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K Reddy. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816*, 2023.

[96] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why we refactor? confessions of github contributors. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*, pages 858–870, 2016.

[97] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 908–911. ACM Press, 2018.

[98] Diomidis Spinellis. *Code quality: the open source perspective.* Adobe Press, 2006.

[99] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. Source code summarization in the era of large language models. *arXiv preprint arXiv:2407.07959*, 2024.

[100] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for software design smells: managing technical debt.* Morgan Kaufmann, 2014.

[101] Balázs Szalontai, Péter Bereczky, and Dániel Horpácsi. Deep learning-based refactoring with formally verified training data. *Infocommunications journal*, 15(SI):2–8, 2023.

[102] Robert Tairas and Jeff Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Information and Software Technology*, 54(12):1297–1307, 2012.

[103] Alexander Trautsch, Johannes Erbel, Steffen Herbold, and Jens Grabowski. What really changes when developers intend to improve their source code: a commit-level study of static metric value and static analysis warning changes. *Empirical Software Engineering*, 28(2):30, 2023.

[104] Nikolaos Tsantalis. Jdeodrant.

[105] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 119–128, 2009.

[106] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of extract method refactoring opportunities for the decomposition of methods. *Journal of Systems and Software*, 84(10):1757–1782, 2011.

[107] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 48(3):930–950, 2022.

[108] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, pages 483–494. ACM, 2018.

[109] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, pages 483–494. ACM, 2018.

[110] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617*, 2020.

[111] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 25–36. IEEE, 2019.

[112] David van der Leij, Jasper Binda, Robbert van Dalen, Pieter Vallen, Yaping Luo, and Maurício Aniche. Data-driven extract method recommendations: A study at ing. page 1337–1347. Association for Computing Machinery, 2021.

[113] David van der Leij, Jasper Binda, Robbert van Dalen, Pieter Vallen, Yaping Luo, and Maurício Aniche. Data-driven extract method recommendations: a study at ing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1337–1347, 2021.

[114] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[115] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[116] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 397–407, 2018.

[117] Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. Compilable neural code generation with compiler feedback. *arXiv preprint arXiv:2203.05132*, 2022.

[118] Xing Wang, Zhaopeng Tu, Longyue Wang, and Shuming Shi. Self-attention with structural position representations. *arXiv preprint arXiv:1909.00383*, 2019.

[119] Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. Rlcoder: Reinforcement learning for repository-level code completion. *arXiv preprint arXiv:2407.19487*, 2024.

[120] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.

[121] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021.

[122] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1282–1294, 2023.

[123] Sihan Xu, Aishwarya Sivaraman, Siau-Cheng Khoo, and Jing Xu. Gems: An extract method refactoring recommender. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 24–34, 2017.

[124] Jinto Yamanaka, Yasuhiro Hayase, and Toshiyuki Amagasa. Recommending extract method refactoring based on confidence of predicted method name. *ArXiv*, abs/2108.11011, 2021.

[125] Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

[126] Xin Yin, Chao Ni, Tien N Nguyen, Shaohua Wang, and Xiaohu Yang. Rectifier: Code translation with corrector via llms. *arXiv preprint arXiv:2407.07472*, 2024.

[127] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. Evaluating and improving chatgpt for unit test generation. 1(FSE), 2024.

[128] Marcelo Serrano Zanetti, Claudio Juan Tessone, Ingo Scholtes, and Frank Schweitzer. Automated software remodularization based on move refactoring: a complex systems approach. In *Proceedings of the 13th international conference on Modularity*, pages 73–84, 2014.

[129] Apostolos V. Zarras, Theofanis Vartziotis, and Panos Vassiliadis. Navigating through the archipelago of refactorings. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 922–925, New York, NY, USA, 2015. Association for Computing Machinery.

[130] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. Large language model for vulnerability detection and repair: Literature review and roadmap. *arXiv preprint arXiv:2404.02525*, 2024.

[131] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

[132] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. Xlcost: A benchmark dataset for cross-lingual code intelligence, 2022.

Appendix A

Complementary Material

This appendix presents the materials and tools used during this thesis.

A.1 Replication Package

The framework, tools, scripts, analysis, and generated data can be found online at:

- https://github.com/SMART-Dal/extract-method-identification
- https://anonymous.4open.science/r/extract-method-generation-2C9B

Appendix B

Copyright Release

B.1 Automatic refactoring candidate identification leveraging effective code representation

Palit, Indranil, Gautam Shetty, Hera Arif, and Tushar Sharma.“Automatic refactoring candidate identification leveraging effective code representation.” In 2023 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 369-374. IEEE, 2023

Following is the copyright agreement with Institute of Electrical and Electronics Engineers.

IEEE COPYRIGHT AND CONSENT FORM

To ensure uniformity of treatment among all contributors, other forms may not be substituted for this form, nor may any wording of the form be changed. This form is intended for original material submitted to the IEEE and must accompany any such material in order to be published by the IEEE. Please read the form carefully and keep a copy for your files.

Automatic Refactoring Candidate Identification Leveraging Effective Code Representation

Indranil Palit, Gautam Shetty, Hera Arif, Tushar Sharma

2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)

COPYRIGHT TRANSFER

The undersigned hereby assigns to The Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE") all rights under copyright that may exist in and to: (a) the Work, including any revised or expanded derivative works submitted to the IEEE by the undersigned based on the Work; and (b) any associated written or multimedia components or other enhancements accompanying the Work.

GENERAL TERMS

1. The undersigned represents that he/she has the power and authority to make and execute this form.
2. The undersigned agrees to indemnify and hold harmless the IEEE from any damage or expense that may arise in the event of a breach of any of the warranties set forth above.
3. The undersigned agrees that publication with IEEE is subject to the policies and procedures of the IEEE PSPB Operations Manual.
4. In the event the above work is not accepted and published by the IEEE or is withdrawn by the author(s) before acceptance by the IEEE, the foregoing copyright transfer shall be null and void. In this case, IEEE will retain a copy of the manuscript for internal administrative/record-keeping purposes.
5. For jointly authored Works, all joint authors should sign, or one of the authors should sign as authorized agent for the others.
6. The author hereby warrants that the Work and Presentation (collectively, the "Materials") are original and that he/she is the author of the Materials. To the extent the Materials incorporate text passages, figures, data or other material from the works of others, the author has obtained any necessary permissions. Where necessary, the author has obtained all third party permissions and consents to grant the license above and has provided copies of such permissions and consents to IEEE

You have indicated that you DO wish to have video/audio recordings made of your conference presentation under terms and conditions set forth in "Consent and Release."

CONSENT AND RELEASE

1. In the event the author makes a presentation based upon the Work at a conference hosted or sponsored in whole or in part by the IEEE, the author, in consideration for his/her participation in the conference, hereby grants the IEEE the unlimited, worldwide, irrevocable permission to use, distribute, publish, license, exhibit, record, digitize, broadcast, reproduce and archive, in any format or medium, whether now known or hereafter developed: (a) his/her presentation and comments at the conference; (b) any written materials or multimedia files used in connection with his/her presentation; and (c) any recorded interviews of him/her (collectively, the "Presentation"). The permission granted includes the transcription and reproduction of the Presentation for inclusion in products sold or distributed by IEEE and live or recorded broadcast of the Presentation during or after the conference.
2. In connection with the permission granted in Section 1, the author hereby grants IEEE the unlimited, worldwide, irrevocable right to use his/her name, picture, likeness, voice and biographical information as part of the advertisement, distribution and sale of products incorporating the Work or Presentation, and releases IEEE from any claim based on right of privacy or publicity.

BY TYPING IN YOUR FULL NAME BELOW AND CLICKING THE SUBMIT BUTTON, YOU CERTIFY THAT SUCH ACTION CONSTITUTES

YOUR ELECTRONIC SIGNATURE TO THIS FORM IN ACCORDANCE WITH UNITED STATES LAW, WHICH AUTHORIZES ELECTRONIC SIGNATURE BY AUTHENTICATED REQUEST FROM A USER OVER THE INTERNET AS A VALID SUBSTITUTE FOR A WRITTEN SIGNATURE.

Indranil Palit	25-08-2023
Signature	**Date (dd-mm-yyyy)**

Information for Authors

AUTHOR RESPONSIBILITIES

The IEEE distributes its technical publications throughout the world and wants to ensure that the material submitted to its publications is properly available to the readership of those publications. Authors must ensure that their Work meets the requirements as stated in section 8.2.1 of the IEEE PSPB Operations Manual, including provisions covering originality, authorship, author responsibilities and author misconduct. More information on IEEE's publishing policies may be found at http://www.ieee.org/publications_standards/publications/rights/authorrightsresponsibilities.html Authors are advised especially of IEEE PSPB Operations Manual section 8.2.1.B12: "It is the responsibility of the authors, not the IEEE, to determine whether disclosure of their material requires the prior consent of other parties and, if so, to obtain it." Authors are also advised of IEEE PSPB Operations Manual section 8.1.1B: "Statements and opinions given in work published by the IEEE are the expression of the authors."

RETAINED RIGHTS/TERMS AND CONDITIONS

- Authors/employers retain all proprietary rights in any process, procedure, or article of manufacture described in the Work.
- Authors/employers may reproduce or authorize others to reproduce the Work, material extracted verbatim from the Work, or derivative works for the author's personal use or for company use, provided that the source and the IEEE copyright notice are indicated, the copies are not used in any way that implies IEEE endorsement of a product or service of any employer, and the copies themselves are not offered for sale.
- Although authors are permitted to re-use all or portions of the Work in other works, this does not include granting third-party requests for reprinting, republishing, or other types of re-use.The IEEE Intellectual Property Rights office must handle all such third-party requests.
- Authors whose work was performed under a grant from a government funding agency are free to fulfill any deposit mandates from that funding agency.

AUTHOR ONLINE USE

- **Personal Servers**. Authors and/or their employers shall have the right to post the accepted version of IEEE-copyrighted articles on their own personal servers or the servers of their institutions or employers without permission from IEEE, provided that the posted version includes a prominently displayed IEEE copyright notice and, when published, a full citation to the original IEEE publication, including a link to the article abstract in IEEE Xplore. Authors shall not post the final, published versions of their papers.
- **Classroom or Internal Training Use.** An author is expressly permitted to post any portion of the accepted version of his/her own IEEE-copyrighted articles on the author's personal web site or the servers of the author's institution or company in connection with the author's teaching, training, or work responsibilities, provided that the appropriate copyright, credit, and reuse notices appear prominently with the posted material. Examples of permitted uses are lecture materials, course packs, e-reserves, conference presentations, or in-house training courses.
- **Electronic Preprints.** Before submitting an article to an IEEE publication, authors frequently post their manuscripts to their own web site, their employer's site, or to another server that invites constructive comment from colleagues. Upon submission of an article to IEEE, an author is required to transfer copyright in the article to IEEE, and the author must update any previously posted version of the article with a prominently displayed IEEE copyright notice. Upon publication of an article by the IEEE, the author must replace any previously posted electronic versions of the article with either (1) the full citation to the IEEE work with a Digital Object Identifier (DOI) or link to the article abstract in IEEE Xplore, or (2) the accepted version only (not the IEEE-published version), including the IEEE copyright notice and full citation, with a link to

the final, published article in IEEE Xplore.

Questions about the submission of the form or manuscript must be sent to the publication's editor.
Please direct all questions about IEEE copyright policy to:
IEEE Intellectual Property Rights Office, copyrights@ieee.org, +1-732-562-3966