Reinforcement Learning vs Supervised Learning: A tug of war to generate refactored code accurately

Indranil Palit
indranil.palit@dal.ca
Dalhousie University
Halifax, Nova Scotia, Canada

Tushar Sharma
tushar@dal.ca
Dalhousie University
Halifax, Nova Scotia, Canada

Abstract

Automated source code refactoring, particularly extract method refactoring, is a crucial and frequently employed technique during software development. Despite its importance and frequent use by practitioners, current automated techniques face significant limitations such as the lack of automation. While machine learning-based approaches have shown promise in intelligent code refactoring, existing such approaches overlook code-specific sequence-level characteristics, including but not limited to compilability, syntactic correctness, and functional integrity. To address these challenges, we propose a novel reinforcement learning-based approach for fine-tuning and aligning code language models to perform automated, intelligent extract method refactoring on Java source code. Our approach fine-tunes sequence-to-sequence generative models and aligns them using the Proximal Policy Optimization (PPO) algorithm using code compilation and presence of the refactoring in the generated code as reward signals. Our experiments demonstrate that our approach significantly enhances the performance of language models in code refactoring. The supervised fine-tuned model, further aligned with PPO, surpasses traditional supervised fine-tuning by 11.96% and 16.45% in terms of BLEU and CodeBLEU scores, respectively. When subjected to a suite of 122 unit tests, the number of successful tests increased from 41 to 66 for the reinforcement learning aligned fine tuned Code-T5 model, highlighting the effectiveness of our approach in producing functionally correct refactorings. Our work paves the way for intelligent, automated code refactoring tools that can significantly reduce developers' manual effort.

CCS Concepts

• **Software and its engineering → Maintaining software**.

Keywords

extract method refactoring, reinforcement learning, large language models

ACM Reference Format:
Indranil Palit and Tushar Sharma. 2025. Reinforcement Learning vs Supervised Learning: A tug of war to generate refactored code accurately. In *Proceedings of The 29th International Conference on Evaluation and Assessment in Software Engineering (EASE 2025).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

EASE 2025, 17–20 June, 2025, Istanbul, Türkiye

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [20, 50]. By removing code smells [20] the practice aims to improve maintainability, encapsulating quality attributes such as readability, flexibility, and testability [10, 16]. Refactoring helps maintain high code quality, facilitating long-term maintainability and evolution [46].

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 [20]. 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 [88]. Extract method refactoring constitutes a significant proportion, approximately 49.6%, of the total refactoring recommendations generated by JDeodorant [71], 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 [66] and industry practitioners [76], underscoring its importance in software maintenance.

Automatically performing extract method refactoring, consist of two major steps [34]. 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 [64] 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 [27] developed a method for generating extract method refactoring candidates using static code analysis tools. Maruyama [27] proposed a candidate generation technique utilizing block-based slicing. Shahidi *et al.* [61] 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 [1, 81]. The field of code generation has seen significant advancements recently, with pre-trained language models such as GitHub Copilot [23] and Amazon Q Developer [7] 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 [12]. Similarly, in our context, current LLM can generate refactored code but often omit the contextual information or generate incomplete, broken, or even uncompilable code [29]. Moreover, using third-party code completion services raises privacy concerns for many organizations. A notable example is Samsung Electronics [28], 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 [2, 68], translation [18, 85], completion [8, 17], bug localization [84], vulnerability detection [43, 89], and program repair [63]. Despite these advancements, to the best of our knowledge, the application of language models for 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.* [55] 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 [12, 29, 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 [72, 73]. We use the dataset to fine-tune four language models, pre-trained on code, using Supervised Fine Tuning (SFT) [26, 31] To enhance model performance and better align it with the objective of generating compilable code while preserving functionality, we use Proximal Policy Optimization (PPO) [60] for reinforcement learning optimization.

Our reinforcement learning approach utilizes an actor-critic architecture [33, 78], 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 [35, 65] 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 paper 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 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. Our replication package including source code and data is available online in our replication package [6].

2 Background

Supervised Fine-Tuning of Large Language Models: Supervised fine-tuning is an add-on training for adapting pre-trained large language models (LLMs), such as CodeT5 [81], 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 [57], 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.

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 [62]. It uses a model known as the Markov Decision Process (MDP) [56], 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. Prior studies [13, 67] has shown that SFT serves as a reliable starting point for RL. Ouyang *et al.* [52] employed a similar two-stage architecture like ours and found that RL performs better when initially fine tuned using SFT. However, none of the works focused on the applicability in the software engineering domain especially in code refactoring.

In the software engineering domain, RL has been used for code completion tasks [39, 65] and code summarization [78]. 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.

In this work, we focus on aligning a fine-tuned large language model for extract method refactoring generation using Proximal Policy Optimization (PPO) [60]—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:

$$V(s) = \mathbb{E}\left[R \mid s\right], where\ R\ is the\ total\ reward \tag{1}$$

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.

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:

$$\mathrm{KL}(\pi_{\theta_{old}} \parallel \pi_\theta) = \sum_x \pi_{\theta_{old}}(x) \log\left(\frac{\pi_{\theta_{old}}(x)}{\pi_\theta(x)}\right), \tag{2}$$

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.

3 Methods

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

3.1 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

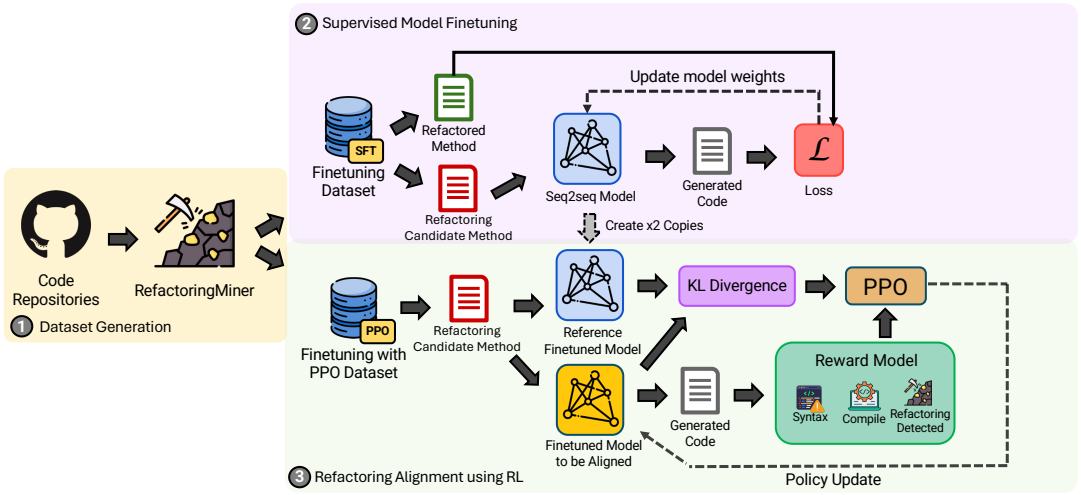


Figure 1: Overview of the proposed approach.

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 PPO with custom reward signals on a fine-tuned model's performance in *extract method* refactoring tasks.

Figure 1 illustrates our methodology. We create our dataset by using the tools such as SEART tool [15] and RefactoringMiner [72, 73]. Following dataset preparation, we fine-tune two encoder-decoder models (code-T5 and PLBART) and two decoder-only models (codeGPT-adapt and codeGen). 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 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.

3.2 Dataset Creation

We employ a systematic approach to identify and collect extract method refactoring instances across multiple open-source Java repositories. Step 1 in Figure 1 shows an overview of the dataset preparation pipeline. We use SEART [15] 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

```
Input: List of repositories R = {r_1, r_2, ..., r.n}
Output: JSONL file with keys "Input" and "Output"
procedure CreateDataset(R)
    Data ← ∅                          ▷ Initialize the dataset as an empty set
    for each repository r_i ∈ R do
        Retrieve branch details for r_i
        Fetch the list of commits for the given branch
        for each commit c_j in the list of commits do
            Identify refactorings performed in c_j
            if extract method refactoring is detected then
                Extract metadata associated with the refactoring
                Extract the refactored method using the metadata
                Checkout to the previous commit c_{j-1}
                Extract the original method from c_{j-1}
                Create output JSON object
                Append this JSON object to Data
            end if
        end for
    end for
    Store Data in a JSONL file
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 [72, 73]. 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 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 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.

3.3 Training Models

3.3.1 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 [80]. 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 [81] 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 [3, 24], code translation [37] and vulnerability detection [25, 54].

PLBART: PLBART [1] 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 [54, 83]

CodeGPT-adapt: codeGPT-adapt [44] 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 [25, 39].

CodeGen: codeGen [49] 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 [14].

Step ❷ in Figure 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.

3.3.2 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 presence of 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.

$$R_{syntax} = \begin{cases} +1 & \text{if no error nodes} \\ -1 & \text{if error nodes present} \end{cases} \tag{3}$$

(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.

$$R_{compile} = \begin{cases} +1 & \text{if code compiles} \\ 0 & \text{if code fails to compile} \end{cases} \tag{4}$$

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

$$R_{detect} = \begin{cases} +1 & \text{if detected by RefactoringMiner} \\ -1 & \text{if not detected} \end{cases} \tag{5}$$

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

$$R_{total} = R_{syntax} + R_{compile} + R_{detect} \tag{6}$$

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 1. 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. 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.

Algorithm 2 DRL Training for Extract Method Refactoring with KL Divergence

Require: Initial policy π_θ, value function V_ϕ, KL divergence coefficient β, weights w_1, w_2, w_3

for each training iteration **do**
  Sample batch of code snippets from dataset
  for each code snippet x **do**
    Generate a refactored snippet using current policy π_θ
    Compute syntactic correctness: R_{syntax} using Eq. 3
    Compute compilation success: $R_{compile}$ using Eq. 4
    Compute refactoring validity: R_{detect} using Eq. 5
    Compute total reward: $R_{total} = w_1 \cdot R_{syntax} + w_2 \cdot R_{compile} + w_3 \cdot R_{detect}$
    Estimate the value of the current state: $V_\phi(s)$
    Calculate advantage: $A = R_{total} - V_\phi(s)$
  end for
  Compute policy update to maximize:
  $J(\theta) = \mathbb{E}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}A\right] - \beta \cdot \mathrm{KL}(\pi_{\theta_{old}}||\pi_\theta)$
  Apply the update to the policy: $\theta_{old} \leftarrow \theta$
  Update value function to minimize: $L(\phi) = \sum(R_{total} - V_\phi(s))^2$
end for

3.3.3 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 [32] 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. A global batch size of 16 is maintained, and the Adam optimizer [32] 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.

3.4 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.

3.4.1 Evaluation Metrics. To assess the effectiveness of our models quantitatively, we utilize established metrics from natural language processing field BLEU [53] and ROUGE [41], as well as specialized metrics tailored for code evaluation, CODEBLEU [59] and syntax match score [90]. The widespread adoption of these metrics in academic research for evaluating generative models supports our decision to use them.

3.4.2 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 3.2. 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. Trivial cases were removed because they do not effectively test the model's ability to handle complex refactoring tasks, providing limited insight into its true capabilities. 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 [51] 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 [47, 74]. We specifically employed the `ChatTester` framework proposed by Yuan *et al.* [86], 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 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 Results

This section summarizes the obtained results corresponding to each research question.

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

The research question aims to evaluate the performance of the fine-tuned models for the refactored code generation. The first part of Table 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.

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 3.4.2, 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 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.

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

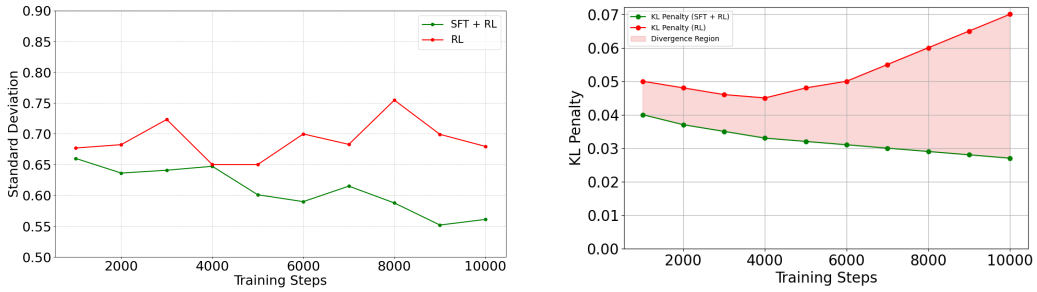
This research question aims to evaluate the application of RL on the refactoring task when applied on pre-trained LLM. The second part of the Table 2 (*i.e.,* PT + RL column) shows the obtained results. Our results show that code-T5 model demonstrates superior performance across all evaluation metrics 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 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.

Qualitatively also, as shown in *RQ2* of Table 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 2. Figure 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 2b reveals an upward trajectory

Table 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



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

Figure 2: RL training observations

Table 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

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.

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

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%. Also, RefactoringMiner identified increased number of cases, from 87 to 98. These results highlight the efficacy of RL in producing accurate extract method refactored code.

Figure 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.

We discuss an illustrative example of an *extract method* refactoring that can be found in our replication package (`README.md` file) [6]. highlighting the differences of SFT and RL techniques. The original method belongs to `aws/aws-dynamodb-encryption-java` repository, 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 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.

> **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.

5 Discussions

Qualitative vs quantitative evaluation: While static 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 2 and Table 3. The evaluation metrics used in Table 2 do not show very drastic difference in the RQ1 and RQ3 results. However, the qualitative results presented in Table 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.

Reward values: In their study of code generation using reinforcement learning, Le *et al.* [38] defined reward values based on heuristic evaluation of functional correctness from unit test signals. We adopted a similar approach for our reward value definition. To validate our reward signal design ($R_{\text{syntax}}, R_{\text{compile}}, R_{\text{detect}}$), we performed a pilot study using a representative subset of $\mathcal{D}_{RL}$. This study evaluated how each reward signal influenced the model's alignment quality. We trained four versions of the PPO-based model: three versions each using one of the reward signals with heuristically chosen values in the range $-1, 0, 1$, and a fourth version incorporating all three signals. The models were evaluated using both quantitative metrics (BLEU, CodeBLEU) and qualitative measures (compilation success rate and refactoring detection rate via RefactoringMiner). Our evaluation demonstrated that the combined reward signal configuration achieved superior performance.

Comparison with baseline: Despite increasing interest in developing fully-automated refactoring code generation tools, we did not find any previous work focusing on automatically generating refactored code using fine-tuned LLM and enhancing it through RL. In our search, we found the study by Szalontai *et al.* [69] as the closest to this study, who addressed a similar problem with a different approach. Their method consists of two stages—code block localization using neural networks, followed by alternative code generation using a Sequence-to-Sequence architecture. Their work employs a grammar-based approach for training data generation. As we could not locate replication package of their study nor could get a response from them, we reconstructed their refactoring generation architecture based on the paper's description and trained it using our dataset $\mathcal{D}_{SFT}$. Our implementation of their approach can be found in our replication package (`/src/baseline/reprod_main.py`). We compared the performance quantitatively, with results shown in Table 4.

Table 4: Comparison with baseline

Approach	BLEU	ROUGE	CodeBLEU
PT + FT + RL (CodeT5)	75.91	79.92	61.87
Baseline Seq-to-Seq	29.77	31.33	19.45

Our approach combining fine-tuning and reinforcement learning significantly outperform the baseline implementation, achieving a 218.1% improvement in CodeBLEU. This substantial performance gap can be attributed to two main factors. First, the baseline architecture uses Bi-LSTM-based model; the model relies on recurrent layers, which struggle with long-term dependencies due to vanishing gradients—a known problem with LSTM-based models. LLMs leverage transformer-based architectures with significantly larger parameter counts, enabling them to capture more complex patterns, long-range dependencies, and contextual relationships in data [9, 77]. Second, the baseline approach do not involve training using real-world refactoring examples, potentially leading to underfitting. RL-aligned LLMs on the other hand, can refine their outputs iteratively to balance trade-offs between fluency, correctness, and task-specific goals [13, 67].

6 Related Work

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.* [5] 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 [22] extended this work by improving the data collection process to create a high-quality, large-scale refactoring dataset. Van Der Leij *et al.* [76] 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. Kurbatova *et al.* [36] employ code embeddings generated from Code2Vec [4] to train their model for Move Method refactoring prediction.

In this domain, researchers have developed a variety of specialized tools and approaches. CeDAR [70], an Eclipse plugin, focuses on identifying and eliminating duplicate code. JDeodorant [45, 71] detects code smells and proposes refactoring strategies. Fokaefs *et al.* [19] extended JDeodorant's capabilities to prioritize and implement class extraction refactorings. SOMOMOTO [87] facilitates

move method refactoring and code modularization. Szalontai *et al.* [69] developed a deep learning method for refactoring source code, initially designed for the Erlang programming language. Tufano *et al.* [75] 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.* [42] proposed RefBERT, a two-stage pre-trained framework based on the BERT architecture.

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

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 [30, 58]. Notably, Ranzato *et al.* [58] 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.* [11], 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.* [79] leveraged compiler feedback as a reward signal to enhance the quality of language model-generated code. Le *et al.* [38] introduced CodeRL, a framework that integrates RL with unit test signals to fine-tune program synthesis models. Shojaee *et al.* [65] 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.* [39], which employs immediate rewards to fine-tune language models for code completion tasks. Wang *et al.* [82] developed RLCoder, combining DRL with Retrieval-Augmented Generation (RAG) pipelines for repository-level code completion. Furthermore, Nichols *et al.* [48] demonstrated the potential of DRL in generating efficient parallel code, expanding the application of these techniques to performance optimization.

To our knowledge, LLMs 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.

7 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.* [39]. Additionally, we employed identical data splits for training and testing across all models, ensuring equitable 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.

8 Conclusions

In this study, we introduce a novel approach that integrates traditional fine-tuning with reinforcement learning alignment to automatically generate extract method refactorings for Java code. To evaluate the generated code, we not only rely on traditional metrics such as BLEU and ROUGE but also construct a detailed qualitative evaluation mechanism to check the syntactic and semantic correctness. Experimental results demonstrate that our approach significantly improves the performance of large language models in code refactoring compared to supervised fine-tuning, as evidenced by quantitative evaluation metrics and qualitative measures.

Our future research will focus on expanding the scope of our approach to encompass various types of refactorings, different programming languages, and industry-based codebases. We also plan to increase the size of our dataset, especially the qualitative evaluation set, for more comprehensive evaluations. We also plan to explore the use of automated test suite generation tools such as EvoSuite [21] to expand our sample size. Furthermore, we intend to investigate alternative reinforcement learning algorithms and reward strategies to further enhance the performance and effectiveness of our automated code refactoring approach.

Code and data availability: Our replication package including source code and data is available online [6].

Acknowledgements

We would like to express our sincere gratitude to Dr. Janarthanan Rajendran for his valuable feedback on the reinforcement learning methodology. We also thank Mootez Saad for his important contribution in preparing figures.

References

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. arXiv:2103.06333 [cs.CL] https://arxiv.org/abs/2103.06333

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

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

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[5] Mauricio Aniche, Erick Maziero, Rafael Durelli, and Vinicius HS Durelli. 2020. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering* 48, 4 (2020), 1432–1450.

[6] Anonymous authors. [n. d.]. Replication package for this study. https://anonymous.4open.science/r/extract-method-generation-2CDC

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

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

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

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

[11] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. 2021. Decision Transformer: Reinforcement Learning via Sequence Modeling. arXiv:2106.01345 [cs.LG] https://arxiv.org/abs/2106.01345

[12] 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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[13] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. Deep reinforcement learning from human preferences. *Advances in neural information processing systems* 30 (2017).

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

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

[16] I Eee. 1990. 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* (1990), 610–12.

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

[18] Hasan Ferit Eniser, Hanliang Zhang, Cristina David, Meng Wang, Brandon Paulsen, Joey Dodds, and Daniel Kroening. 2024. Towards Translating Real-World Code with LLMs: A Study of Translating to Rust. *arXiv preprint arXiv:2405.11514* (2024).

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

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

[21] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. ACM, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[22] Jan Gerling. 2020. Machine learning for software refactoring: a large-scale empirical study. (2020).

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

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

[25] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. arXiv:2308.10620 [cs.SE] https://arxiv.org/abs/2308.10620

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

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

[28] Contributing Writer Jai Vijayan. 2023. Samsung engineers feed sensitive data to CHATGPT, sparking workplace AI warnings. https://www.darkreading.com/vulnerabilities-threats/samsung-engineers-sensitive-data-chatgpt-warnings-ai-use-workplace

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

[30] Yaser Keneshloo, Tian Shi, Naren Ramakrishnan, and Chandan K. Reddy. 2019. Deep Reinforcement Learning For Sequence to Sequence Models. arXiv:1805.09461 [cs.LG] https://arxiv.org/abs/1805.09461

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

[32] Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG] https://arxiv.org/abs/1412.6980

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

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

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

[36] Zarina Kurbatova, Ivan Veselov, Yaroslav Golubev, and Timofey Bryksin. 2020. Recommendation of move method refactoring using path-based representation of code. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 315–322.

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

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

[39] Bolun Li, Zhihong Sun, Tao Huang, Hongyu Zhang, Yao Wan, Ge Li, Zhi Jin, and Chen Lyu. 2024. IRCoCo: Immediate Rewards-Guided Deep Reinforcement Learning for Code Completion. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 182–203.

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

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

[42] Hao Liu, Yanlin Wang, Zhao Wei, Yong Xu, Juhong Wang, Hui Li, and Rongrong Ji. 2023. 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*. 740–752.

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

[44] 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. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. arXiv:2102.04664 [cs.SE] https://arxiv.org/abs/2102.04664

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

[46] Raimund Moser, Pekka Abrahamsson, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. 2007. 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*. Springer, 252–266.

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

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

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

[50] William F. Opdyke. 1992. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. Ph. D. Dissertation. University of Illinois at Urbana-Champaign.

[51] Open-AI. [n. d.]. OpenAI API. https://platform.openai.com/docs/models/gpt-4o [Accessed 19-01-2025].

[52] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. arXiv:2203.02155 [cs.CL] https://arxiv.org/abs/2203.02155

[53] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* (Philadelphia, Pennsylvania) *(ACL '02)*. Association for Computational Linguistics, USA, 311–318. https://doi.org/10.3115/1073083.1073135

[54] Rishov Paul, Md. Mohib Hossain, Mohammed Latif Siddiq, Masum Hasan, Anindya Iqbal, and Joanna C. S. Santos. 2023. Enhancing Automated Program Repair through Fine-tuning and Prompt Engineering. arXiv:2304.07840 [cs.LG] https://arxiv.org/abs/2304.07840

[55] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Andrey Sokolov, Timofey Bryksin, and Danny Dig. 2024. EM-Assist: Safe Automated ExtractMethod Refactoring with LLMs. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE '24)*. ACM. https://doi.org/10.1145/3663529.3663803

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

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

[58] Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2016. Sequence Level Training with Recurrent Neural Networks. arXiv:1511.06732 [cs.LG] https://arxiv.org/abs/1511.06732

[59] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE] https://arxiv.org/abs/2009.10297

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

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

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

[63] Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, Indira Vats, Hadi Moazen, and Federica Sarro. 2024. A survey on machine learning techniques applied to source code. *Journal of Systems and Software* 209 (2024), 111934. https://doi.org/10.1016/j.jss.2023.111934

[64] Tushar Sharma and Diomidis Spinellis. 2018. A survey on software smells. *Journal of Systems and Software* 138 (2018), 158 – 173. https://doi.org/10.1016/j.jss.2017.12.034

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

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

[67] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. 2020. Learning to summarize with human feedback. *Advances in Neural Information Processing Systems* 33 (2020), 3008–3021.

[68] Weisong Sun, Yun Miao, Yuekang Li, Hongyu Zhang, Chunrong Fang, Yi Liu, Gelei Deng, Yang Liu, and Zhenyu Chen. 2024. Source Code Summarization in the Era of Large Language Models. *arXiv preprint arXiv:2407.07959* (2024).

[69] Balázs Szalontai, Péter Bereczky, and Dániel Horpácsi. 2023. Deep Learning-Based Refactoring with Formally Verified Training Data. *Infocommunications journal* 15, SI (2023), 2–8.

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

[71] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of Extract Method Refactoring Opportunities. In *2009 13th European Conference on Software Maintenance and Reengineering*. 119–128. https://doi.org/10.1109/CSMR.2009.23

[72] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2022), 930–950. https://doi.org/10.1109/TSE.2020.3007722

[73] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/3180155.3180206

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

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

[76] David van der Leij, Jasper Binda, Robbert van Dalen, Pieter Vallen, Yaping Luo, and Maurício Aniche. 2021. 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*. 1337–1347.

[77] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).

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

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

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

[81] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. arXiv:2109.00859 [cs.CL] https://arxiv.org/abs/2109.00859

[82] Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. RLCoder: Reinforcement Learning for Repository-Level Code Completion. *arXiv preprint arXiv:2407.19487* (2024).

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

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

[85] Xin Yin, Chao Ni, Tien N Nguyen, Shaohua Wang, and Xiaohu Yang. 2024. Rectifier: Code Translation with Corrector via LLMs. *arXiv preprint arXiv:2407.07472* (2024).

[86] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and Improving ChatGPT for Unit Test Generation. 1, FSE (2024). https://doi.org/10.1145/3660783

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

[88] Apostolos V. Zarras, Theofanis Vartziotis, and Panos Vassiliadis. 2015. Navigating through the archipelago of refactorings. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 922–925. https://doi.org/10.1145/2786805.2803203

[89] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2024. Large Language Model for Vulnerability Detection and Repair: Literature Review and Roadmap. *arXiv preprint arXiv:2404.02525* (2024).

[90] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. 2022. XLCoST: A Benchmark Dataset for Cross-lingual Code Intelligence. arXiv:2206.08474 [cs.SE] https://arxiv.org/abs/2206.08474