A Comprehensive Evaluation of Parameter-Efficient Fine-Tuning on Method-Level Code Smell Detection

BEIQI ZHANG, School of Computer Science, Wuhan University, China
PENG LIANG, School of Computer Science, Wuhan University, China
XIN ZHOU, School of Computing and Information Systems, Singapore Management University, Singapore
XIYU ZHOU, School of Computer Science, Wuhan University, China
DAVID LO, School of Computing and Information Systems, Singapore Management University, Singapore
QIONG FENG, School of Computer Science, Nanjing University of Science and Technology, China
ZENGYANG LI, School of Computer Science, Central China Normal University, China
LIN LI, School of Computer Science and Artificial Intelligence, Wuhan University of Technology, China

Code smells, which are suboptimal coding practices that can potentially lead to defects or maintenance issues, can negatively impact the quality of software systems. Most existing code smell detection methods rely on heuristics-based or machine learning (ML) and deep learning (DL)-based techniques. However, these techniques have several drawbacks (e.g., unsatisfactory performance). Large Language Models (LLMs) have garnered significant attention in the software engineering (SE) field, achieving state-of-the-art performance across a wide range of SE tasks. Parameter-Efficient Fine-Tuning (PEFT) methods, which are commonly used to adapt LLMs to specific tasks with fewer parameters and reduced computational resources, have emerged as a promising approach for enhancing the performance of LLMs in various SE tasks. However, LLMs have not yet been explored for code smell detection, and their effectiveness for this task remains unclear. Furthermore, no comprehensive investigation has been conducted on the efficiency of PEFT methods for method-level code smell detection. In this regard, we systematically evaluate the effectiveness of state-of-the-art PEFT methods on both small and large Language Models (LMs) for method-level code smell detection. To begin, we constructed high-quality Java code smell datasets sourced from GitHub. We then fine-tuned four small LMs and six LLMs using various PEFT techniques, including prompt tuning, prefix tuning, LoRA, and $(IA)^3$, for code smell detection. Our comparison against full fine-tuning revealed that PEFT methods not only achieve comparable or better effectiveness but also consume less peak GPU memory. Our analysis further explored the performance of small LMs versus LLMs in the context of code smell detection. Surprisingly, we found that LLMs did not outperform small LMs in this specific task, suggesting that smaller models may be more suited for method-level code smell detection. We also investigated the impact of varying hyper-parameter settings and the availability of training data. Our results show that increasing the number of trainable parameters does not necessarily improve the effectiveness of PEFT methods for code smell detection. In contrast, we

Authors' addresses: Beiqi Zhang, zhangbeiqi@whu.edu.cn, School of Computer Science, Wuhan University, Wuhan, China; Peng Liang, liangp@whu.edu.cn, School of Computer Science, Wuhan University, Wuhan, China; Xin Zhou, xinzhou.2020@phdcs.smu.edu.sg, School of Computing and Information Systems, Singapore Management University, Singapore; Xiyu Zhou, xiyuzhou@whu.edu.cn, School of Computer Science, Wuhan University, Wuhan, China; David Lo, davidlo@smu.edu.sg, School of Computing and Information Systems, Singapore Management University, Singapore; Qiong Feng, qiongfeng@njust.edu.cn, School of Computer Science, Nanjing University of Science and Technology, Nanjing, China; Zengyang Li, zengyangli@ccnu.edu.cn, School of Computer Science, Central China Normal University, Wuhan, China; Lin Li, cathylilin@whut.edu.cn, School of Computer Science and Artificial Intelligence, Wuhan University of Technology, Wuhan, China.

1049-331X/2024/0-ART0 $15.00
https://doi.org/XXXXXXX.XXXXXXX

observed significant performance gains when the training dataset size was increased. Finally, our results show that LMs combined with PEFT methods outperform traditional heuristic-based detectors, advanced DL-based approaches, and GPT-3.5-turbo in method-level code smell detection. Specifically, MCC values improve by 13.24% to 50.59% for *Complex Conditional* and 11.22% to 54.32% for *Complex Method*, demonstrating a more effective solution for identifying these code smells. In conclusion, our work provides a comprehensive evaluation of PEFT methods on both small and large LMs for method-level code smell detection. The results highlight the efficiency and effectiveness of PEFT methods in this domain, offering valuable insights for optimizing LMs for method-level code smell detection. We suggest that the choice of PEFT method for code smell detection should be guided by the model used, data availability, and computational resources. Our replication package is available at [1], enabling others to reproduce and extend our findings.

CCS Concepts: • **Software and its engineering → Software maintenance tools**.

Additional Key Words and Phrases: Code Smell Detection, Large Language Model, Parameter-Efficient Fine-Tuning

ACM Reference Format:
Beiqi Zhang, Peng Liang, Xin Zhou, Xiyu Zhou, David Lo, Qiong Feng, Zengyang Li, and Lin Li. 2024. A Comprehensive Evaluation of Parameter-Efficient Fine-Tuning on Method-Level Code Smell Detection. *ACM Trans. Softw. Eng. Methodol.* 0, 0, Article 0 (2024), 30 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Code smells are indicators of potential issues in code or design [2], which could adversely affect software quality, particularly in terms of maintainability [3], code readability [4], and testability [5]. Detecting code smells can be used to identify software design issues that lead to potential problems in future development and maintenance [2, 6, 7]. Manual identification of code smells can be time-consuming and costly [8], prompting many researchers to explore automated methods for more efficient code smell detection. Existing automated code smell detection methods are typically heuristics-based [7, 9, 10]. Traditional heuristics-based detectors for code smells rely on a set of manually curated features and thresholds to discriminate between smelly and non-smelly classes [11]. While it has been shown that these static detectors provide reasonably accurate results, they are proven to have limitations that hinder their practical application [12, 13], particularly because their performance is highly dependent on the selection of thresholds. Besides, code smells identified by such heuristics-based methods can be subjectively interpreted by developers, leading to a significant gap between detection results and developers' agreement [11, 14–16].

To address these issues, many researchers have incorporated machine learning (ML) and deep learning (DL) algorithms to mitigate the performance instability in code smell detection that arises from the manual threshold-setting process in traditional detectors [17–20]. By learning to discern between smelly and non-smelly code elements, ML and DL techniques offer a solution to the subjectivity problem inherent in heuristics-based methods [21]. However, despite these advancements, ML and DL methods still face significant challenges in achieving considerable performance for code smell detection [22, 23].

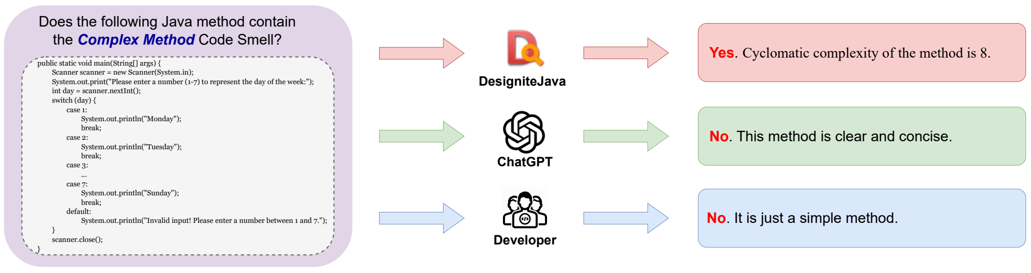
In recent years, in the field of Natural Language Processing (NLP), increased computational power, advanced machine learning techniques, and access to very large-scale data have led to the emergence of Large Language Models (LLMs). The exceptional performance of LLMs in NLP tasks (e.g., text classification, generation, and summarization) has garnered significant attention and interest from the Software Engineering (SE) domain. Researchers have begun employed LLMs, particularly code-specific LLMs like CodeBERT-Base [24] and CodeT5 [25], using fine-tuning techniques to tackle various SE tasks, such as code review [26] and vulnerability repair [27]. Among all the fine-tuning methods, Parameter-Efficient Fine-Tuning (PEFT) has emerged as a promising approach, offering an efficient alternative to full fine-tuning, particularly when computational resources are limited. PEFT methods have shown potential in adapting LLMs to various downstream tasks with fewer parameters and reduced resource requirements, thus expanding the practical applicability of LLMs in diverse SE contexts. Previous studies have explored the efficiency of PEFT methods for Language Models (LMs) on automated program repair [28], code generation [29], code search [30], and etc. However, to the best of our knowledge, there is currently no comprehensive research exploring the use of LLMs for code smell detection, nor any in-depth studies analyzing the application of PEFT methods in this area.

Moreover, utilizing PEFT methods for code smell detection can facilitate just-in-time code smell detection. Traditional heuristic-based methods for code smell detection are often applied after the software systems have been completed [31–33]. As a result, code smells are typically discovered at later stages of development, requiring refactoring that involves modifying all relevant parts of the system, thereby increasing the cost and complexity of addressing these code smells. In contrast, incorporating LLMs with PEFT techniques allows for real-time detection of code smells, enabling developers to identify and address potential code smells as they write code. For instance, upon completing a function, developers can immediately receive feedback on whether it contains a code smell. This proactive approach contributes to maintaining high code quality throughout the development life cycle, leading to more efficient and effective code maintenance and improvement [34, 35].

To this end, we conduct a series of experiments on fine-tuning both small and large LMs with PEFT methods on the code smell detection task. In this paper, we define small LMs as those with fewer than 1B parameters, while LMs with 1B parameters or more are considered large LMs. Specifically, we construct high-quality Java code smell datasets comprising two types of widely studied and sophisticated method-level code smells, i.e., *Complex Conditional* (*CC*) and *Complex Method* (*CM*), as no existing datasets are available. We then apply various PEFT methods, including prompt tuning, prefix tuning, LoRA, and $(IA)^3$, to fine-tune four popular small LMs and six cutting-edge LLMs with parameter sizes ranging from 125M to 7B. The effectiveness of these PEFT methods is assessed for code smell detection, and their performance is compared to that of full fine-tuning in terms of both effectiveness and peak GPU memory usage. Additionally, we investigate the performance of PEFT techniques under different hyper-parameter settings and evaluate their effectiveness in low-resource scenarios with varying numbers of training samples on the method-level code smell detection task. Finally, we employ a heuristics-based static analysis tool alongside two state-of-the-art DL-based code smell detection tools to compare the performance of PEFT methods with existing approaches.

The **main contributions** of this work are as follows:

- We mine Java repositories from GitHub and employ a traditional heuristic-based detector, along with manual validation, to create high-quality datasets for two types of method-level code smells, i.e., *Complex Conditional* and *Complex Method*.
- We conduct a comprehensive evaluation of multiple PEFT methods on both small and large LMs for code smell detection. Our results indicate that LLMs, when fine-tuned, do not always outperform small LMs. Additionally, PEFT methods achieve better or comparable effectiveness compared to full fine-tuning.
- We explore the performance of PEFT methods under different hyper-parameter settings and in low-resource scenarios. Our results reveal that adjusting hyper-parameters to increase trainable parameters does not necessarily lead to better effectiveness in detecting code smells. However, the performance of PEFT methods improves with more training samples.
- We compare a heuristic-based detector and two best-in-class DL-based code smell detection tools with both small and large LMs using PEFT methods on code smell detection. Our results


Does the following Java method contain the *Complex Method* Code Smell?

```
public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Please enter a number (1-7) to represent the day of the week:");
        int day = scanner.nextInt();
        switch (day) {
                case 1:
                        System.out.println("Monday");
                        break;
                case 2:
                        System.out.println("Tuesday");
                        break;
                case 3:
                        ...
                case 7:
                        System.out.println("Sunday");
                        break;
                default:
                        System.out.println("Invalid input! Please enter a number between 1 and 7.");
        }
        scanner.close();
}
```

Yes. Cyclomatic complexity of the method is 8.

No. This method is clear and concise.

No. It is just a simple method.

Fig. 1. An example of a Java method that is considered to contain *Complex Method* code smell by the heuristic-based tool

demonstrate that both small and large LMs with PEFT methods consistently outperform the existing heuristic- and DL-based approaches in detecting the method-level code smells, highlighting the superiority of PEFT methods.

- We provide actionable suggestions and insights for both developers and researchers for using PEFT methods on method-level code smell detection.

The remainder of this paper is organized as follows: Section 2 presents the background information. Section 3 outlines the research questions and the process of dataset construction, while the study results are provided in Section 4. Section 5 discusses the insights for practitioners. Potential threats to validity are addressed in Section 6, with related work reviewed in Section 7. Finally, Section 8 concludes the paper and suggests future directions for research.

2 BACKGROUND

2.1 The Need of Applying Language Models on Code Smell Detection

Existing heuristic-based code smell detectors rely on pre-defined rules and metrics, while ML- or DL-based approaches depend on algorithms to identify the optimal threshold for code smell classification. However, these tools often fail to consider the inherent context of the code, which is crucial for accurately detecting and understanding code smells [19]. For example, the Java method shown in Figure 1 contains a switch-case statement with seven cases. From the perspective of a Java developer, this method simply maps the numbers 1 through 7 to the days of the week, from Monday to Sunday. It is straight-forward and easy to understand, as all the case statements follow the same structure. However, when scanned by an existing heuristic-based code smell detector, such as DesigniteJava [31], the tool flags the method with the *Complex Method* code smell. This decision is based solely on the method's cyclomatic complexity, which exceeds a pre-defined threshold, without considering the method's context.

In contrast, Language Models (LMs), such as ChatGPT [36], have the advantage of considering the inherent context of the code [25, 37]. The contextual understanding allows the LM to accurately determine that the method does not exhibit *Complex Method*, as it can perform a role similar to a skilled developer assessing the code. The LM can evaluate not only traditional metrics, such as cyclomatic complexity, but also analyze additional factors, including readability, modularity, and overall design. Therefore, compared to traditional heuristic-based static analysis tools and certain ML- or DL-based detectors, LMs have the potential to detect code smells more accurately by considering both structural and contextual factors, rather than relying solely on numerical thresholds or pre-defined rules.

2.2 Full Fine-tuning

Language Models (LMs) which are pre-trained on massive general-purpose corpus via self-supervised method, can be applied to various downstream NLP tasks by adapting their rich possession of learned knowledge through full fine-tuning [38]. Full fine-tuning involves updating all parameters of an LM, including multi-head attention (MHA), feed-forward neural network (FFN), layer normalization, and so on. However, as LMs scale up, full fine-tuning a large-sized LM, i.e., LLM, can be excessively costly and ultimately make it practically unfeasible [39, 40].

2.3 Parameter-Efficient Fine-Tuning (PEFT) Methods

Instead of full fine-tuning, pre-trained LMs could be tailored to downstream tasks by Parameter-Efficient Fine-Tuning (PEFT) methods. PEFT methods involve optimizing only a subset of model parameters or incorporating external modules for new tasks while keeping the majority of parameters frozen. Compared to full fine-tuning, PEFT methods can reduce the substantial computational and data demands of LLMs and achieve comparable or even better results [41]. The right side of Figure 2 illustrates several state-of-the-art PEFT methods employed in this study, including prompt tuning, prefix tuning, LoRA, and $(IA)^3$.

Prompt tuning [42] directly appends learnable vectors, known as soft prompts, to the initial input embedding layer. When fine-tuning PLMs with prompt tuning, only the soft prompts are updated, while the remaining model weights and architecture are kept fixed (not updated).

Prefix tuning [43] introduces adjustable prefix (i.e., a sequence of task-specific vectors) across all Transformer layers. Specifically, these trainable prefix vectors are prepended to the key K and value V of all the MHA mechanisms. Prefix tuning adopts a reparameterization strategy, utilizing a multilayer perceptron (MLP) layer to generate the added vectors instead of optimizing them in a straightforward manner [41].

LoRA (Low-Rank Adaptation) [44] injects trainable low-rank decomposition matrices into the query W_q, key W_k, and value W_v projection matrices of the MHA. For a pre-trained weight matrix $W_0 \in \{W_q, W_k, W_v\}$, an additive operation $W_0 + \Delta W = W_0 + BA$ is performed, where the rank of matrices B and A is considerably smaller than that of W_0, consequently effectively reducing the number of trainable parameters while preserving the essential characteristics of the original matrices.

(IA)^3 [45] (Infused Adapter by Inhibiting and Amplifying Inner Activation) incorporates three learnable rescaling vectors, l_k, l_v, and l_{ff}, to rescale the key K and value V in MHA and the inner activations in FFN. The operations conducted within the MHA can be outlined as follows:

$$\text{MHA(x)} = Softmax\left(\frac{Q(l_k \odot K^T)}{\sqrt{d_k}}\right)(l_v \odot V) \tag{1}$$

where $\odot$ denotes the element-wise multiplication and d_k refers to the dimension of key K.

The modifications in the FFN can be defined as:

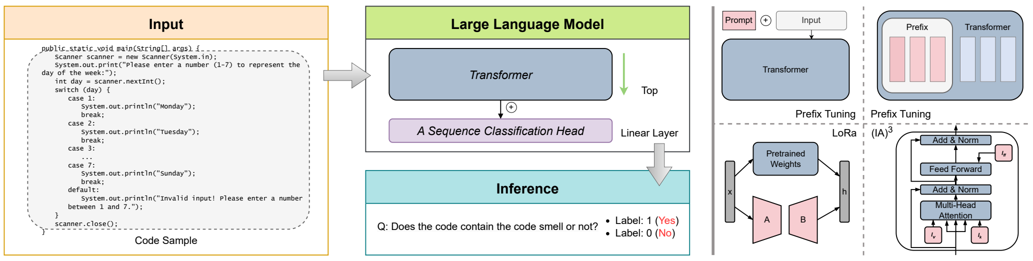
$$\text{FFN(x)} = W_{up}(l_{ff} \odot \sigma(W_{down}x)) \tag{2}$$

where σ is the activation function within FFN, and W_{up} and W_{down} indicate the weight matrices of the FFN.

3 METHODOLOGY

3.1 Research Goal

Figure 2 illustrates the architecture of our LLM-based code smell detector using PEFT methods. We frame the method-level code smell detection task as a binary classification problem, where



```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Please enter a number (1-7) to represent the
day of the week:");
    int day = scanner.nextInt();
    switch (day) {
        case 1:
            System.out.println("Monday");
            break;
        case 2:
            System.out.println("Tuesday");
            break;
        case 3:
            ...
        case 7:
            System.out.println("Sunday");
            break;
        default:
            System.out.println("Invalid input! Please enter a number
        between 1 and 7.");
    }
    scanner.close();
}
```

Q: Does the code contain the code smell or not?

- Label: 1 (Yes)
- Label: 0 (No)

Fig. 2. Left: architecture of our proposed method for method-level code smell detection. Right: a detailed illustration of the PEFT module structure for the four PEFT methods.

the input is a piece of code. A sequence classification head is added on top of the LLM to handle the classification task. For model fine-tuning, we use four distinct PEFT methods. The model then outputs either 0 or 1, representing the absence or presence of the specific type of code smell.

3.2 Research Questions

This study aims to evaluate the effectiveness of state-of-the-art PEFT methods (prompt tuning, prefix tuning, LoRA, and $(IA)^3$) in small LMs and LLMs for method-level code smell detection. To achieve this goal, we conducted a set of experiments to answer the following Research Questions (RQs):

RQ1: What is the effectiveness of different PEFT methods and full fine-tuning on various small LMs and LLMs? Previous studies have shown remarkable results using PEFT methods to fine-tune small LMs for software engineering tasks like code clone detection and defect detection [28–30, 46, 47]. However, these studies have not paid attention to the effectiveness of PEFT methods for code smell detection. With this RQ, we aim to compare and validate the effectiveness of PEFT methods across multiple LMs of varying parameter sizes (see Section 3.4) specifically for code smell detection.

RQ2: What is the effectiveness of PEFT methods with different hyper-parameters? When fine-tuning LMs for downstream tasks, the configuration of hyper-parameters plays a crucial role in determining the final results by affecting the number of trainable parameters. In prompt tuning and prefix tuning, the number of virtual tokens serves as a critical hyper-parameter [42, 43]. For LoRA, the key hyper-parameter is the rank r [44], while in $(IA)^3$, it is the selection of activated modules [45]. Due to our limited computational resources and the popularity of LoRA compared to other PEFT methods [48, 49], we decided to employ LoRA to fine-tune a small LM and an LLM with different r values in a range of {8, 32, 64, 640, 1,280, 2,560} to explore the effect of hyper-parameter settings on PEFT methods for code smell detection.

RQ3: What is the effectiveness of PEFT methods in low-resource scenarios? Low-resource scenarios, characterized by constraints on the amount of training data, pose significant challenges to fine-tuning LMs for specific tasks. Due to the scarcity of real-world datasets [50], these scenarios are common in tasks such as code smell detection. Therefore, it is crucial to investigate how the number of training samples influences the effectiveness of PEFT methods for code smell detection.

RQ4: What is the effectiveness of PEFT methods compared to state-of-the-art code smell detection approaches? Existing code smell detectors include traditional heuristic-based and DL-based methods. However, heuristic-based approaches for code smell detection depend on rules and heuristics derived from software metrics [7, 51–53], which may suffer from several limitations (e.g. different tools may produce inconsistent results, complicating developers' refactoring decisions

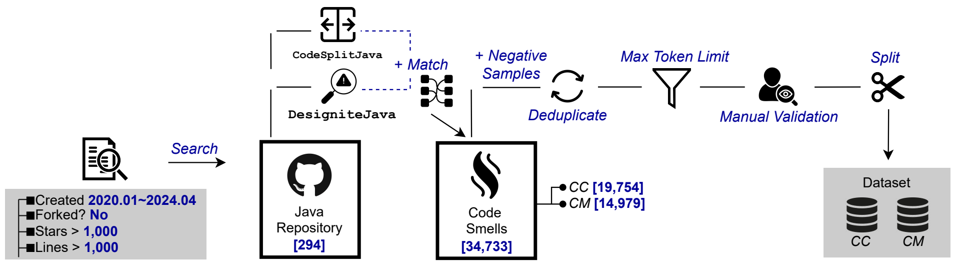


Fig. 3. Overview of dataset construction

[54]). As a result, researchers have been exploring the application of DL techniques as alternatives for code smell detection due to the drawbacks of heuristic-based methods [55], and state-of-the-art DL-based code smell detectors have demonstrated strong effectiveness in identifying code smells [19, 23]. With this RQ, we aim to assess how LMs utilizing PEFT methods compare against heuristic-based code smell detectors and established DL techniques, including general-purpose LLMs, in terms of their ability to detect and classify code smells accurately.

3.3 Dataset Construction

In this section, we elaborate on the process with the following steps of constructing the high-quality datasets for two types of method-level code smells used to fine-tune both small and large LMs for code smell detection. We created our own datasets because most existing datasets only offer code metrics related to the code smells instead of code snippets, which are not usable for our purposes [56–58]. While some code smell datasets provide valid code samples, they do not include *Complex Conditional* and *Complex Method* [59], two types of method-level code smells. Additionally, the source projects from these existing datasets are quite outdated [60–63]. Considering this, we aim to construct new datasets sourced from the latest trending GitHub projects. Figure 3 illustrates the overview process of dataset construction.

3.3.1 Selected Code Smells. In this study, we intend to use LLMs to detect two types of method-level code smells, i.e., *Complex Conditional* and *Complex Method*. The definitions of the two types of code smells are provided below:

- ***Complex Conditional (CC)*:** This code smell occurs when a conditional statement is excessively intricate [64].
- ***Complex Method (CM)*:** This code smell occurs when a method exhibits high cyclomatic complexity [64].

We chose *CC* and *CM* because they have been widely studied in research on DL-based code smell detection (e.g., [19, 23, 65, 66]). Another reason for selecting *CC* and *CM* is the availability of detection tools, such as DesigniteJava [31], which can assist us in collecting candidate code samples that may exhibit these two types of code smells. We excluded very simple smells (e.g., *Long Method*), which can be easily identified by basic techniques. In contrast, detecting the two selected method-level code smells requires LLMs to spot specific features, pay attention to cohesion and coupling aspects, and analyze the entire method for its structural properties [19], thereby expanding the ambition and enhance the applicability of this study.

3.3.2 Java Repository Selection. We used GitHub REST API[1] to search Java repositories from GitHub. From the extensive array of available Java projects hosted on GitHub, we set the following criteria:

- The repositories should be created between January 1, 2020 and April 30, 2024.
- The repositories should exclude forked ones to avoid duplicates.
- The repositories should have a minimum of 1,000 stars and 1,000 lines of code.

These criteria ensured that the selected Java repositories were recent, popular, active, and non-trivial Java repositories. By employing the stringent filtering criteria, we obtained a list of 294 GitHub repositories coded in Java.

3.3.3 Potential Code Smell Detection. DesigniteJava is an open-source tool that supports detection of 18 design and 10 implementation smells, which have been extensively used in many studies as a mature code smell detector [56, 67, 68]. In this paper, we had developed DesigniteJava to detect potential *CC* and *CM* in the Java repositories we collected in Section 3.3.2. After this step, we got a list of code smells, consisting of the project name, package name, class name, and method name where each detected code smell occurs. It is important to note that the code smell detection results from DesigniteJava may not be entirely accurate, as the detection results can contain both false positives and false negatives. False positives occur when the tool incorrectly flags code as exhibiting *CC* or *CM*, while false negatives happen when actual code smells are missed. These inaccuracies stem from the rule-based and heuristic nature of DesigniteJava, which relies on predefined criteria that may not capture all nuances of the code.

3.3.4 Data Preparation. CodeSplitJava [69] is a utility program designed to partition methods from Java source code into distinct files. We used CodeSplitJava to parse the 294 Java repositories collected in Section 3.3.2 into code files. Each of these code files corresponds to a method in the 294 repositories, and its path reflects the hierarchical structure of the method (i.e., namespaces and packages are converted into folders). By correlating the list of detected code smells collected in Section 3.3.3, with the paths of the code files generated by CodeSplitJava, we identified the method-level code snippets with code smells and those without (i.e., *Clean Code*).

We removed any duplicate samples within each type of code smell to ensure uniqueness. Additionally, we limited the maximum token length of the samples to 1,024, filtering out any that exceeded this limit. Table 1 provides the number of potential positive samples after each step for the two types of code smells. In total, we collected 4,266 potential positive code samples containing *CC* and 6,479 containing *CM*. For both *CC* and *CM*, we introduced an equal number of potential negative samples (i.e., code samples without code smells as identified by DesigniteJava) for the subsequent steps. We also verified that all negative samples adhered to the maximum token limit and were free of duplicates. However, the quality of these datasets may be compromised as they could contain false positives and false negatives, requiring further refinement.

3.3.5 Manual Validation. After the above steps, we obtained an equal number of potential positive and negative samples for both *CC* ($4,266 \times 2 = 8,532$) and *CM* ($6,479 \times 2 = 12,958$). To ensure the quality of the datasets for these two types of code smells and to exclude false positives and false negatives, a manual validation was conducted by the first and fourth authors. Initially, a pilot validation was performed at the 95% confidence level, with a margin of error of less than 5%, resulting in the random selection of 368 samples from *CC* and 374 from *CM*. The first and the fourth author annotated the selected samples independently, and the pilot annotations were subsequently compared. Cohen's Kappa coefficient [70] was used to measure the inter-rater agreement for both

[1] https://docs.github.com/en/rest?apiVersion=2022-11-28

Table 1. Number of potential positive code samples after each step of data preparation (**#Initial Samples** shows the initially collected potential positive samples, **#After Deduplication** reflects the number of samples after duplicate removal, and **#After Token Limit** indicates the count of samples that pass the token length filter, i.e., no more than 1,024 tokens)

Code Smell	**#Initial Samples**	**#After Deduplication**	**#After Token Limit**
Complex Conditional	19,754	4,589	4,266
Complex Method	14,979	9,781	6,479

CC and *CM*. The resulting Kappa coefficient for *CC* was 0.85, while for *CM* it was 0.81, both exceeding 0.7, indicating a strong level of agreement between the two annotators. Following the pilot validation, the two authors manually annotated the remaining code samples. Throughout the entire validation process, including the pilot phase, any disagreements between the first and fourth authors were resolved through discussions involving the second author, ensuring consistency in the annotations. Overall, we identified 2,708 false positives and 29 false negatives for *CC*, and 2,376 false positives and 20 false negatives for *CM*. The numbers of both positive and negative samples for *CC* and *CM* after manual validation are presented in Table 2.

Table 2. Number of positive and negative samples after data annotation

Code Smell	**#Positive**	**#Negative**	**Total**
Complex Conditional	1,587	6,945	8,532
Complex Method	4,123	8,835	12,958

3.3.6 Data Split. After manually reviewing the entire dataset, we split the code samples for both *CC* and *CM* into training, validation, and test sets following an 8:1:1 ratio, as employed in previous studies [71, 72]. Specifically, the positive and negative samples for each method-level code smell were distributed according to the same 8:1:1 proportion. This approach ensures a consistent distribution of positive and negative cases, providing a solid foundation for model training, validation, and evaluation. The number of code samples in the datasets of *CC* and *CM* after this step is presented in Table 3. We used these clean and high-quality datasets as our final datasets for a series of experiments.

Table 3. Number of positive and negative samples after data split

Set	**Complex Conditional**			**Complex Method**		
	#Positive	**#Negative**	**#Total**	**#Positive**	**#Negative**	**#Total**
Train	1,269	5,557	6,826	3,299	7,067	10,366
Valid	159	694	853	412	884	1,296
Test	159	694	853	412	884	1,296
Total	1,587	6,945	8,532	4,123	8,835	12,958

3.4 Base Model Selection

In this work, we aim to explore the effectiveness of PEFT methods on method-level code smell detection task. To ensure a thorough analysis, we have chosen to use both small LMs and LLMs as our base models. Note that we deem models with 1B or more parameters as LLMs, while referring to those with fewer parameters as small language models.

First, the selected LMs should be open-source as PEFT methods require access to their parameters. Second, the models should support classification tasks as we frame code smell detection as a binary classification problem. Third, the base models used in this study must be able to be fine-tuned and tested on a single 48GB GPU (our maximum computational resource) without encountering memory overflow. Fourth, the selected models should either be popular small LMs or recent advanced LLMs to ensure they represent state-of-the-art capabilities.

Based on the above criteria, we employed 4 small LMs (i.e., CodeBERT [24], GraphCodeBERT [73], UnixCoder [74], and CodeT5 [25]) and 6 LLMs (DeepSeek-Coder-1.3B [75], DeepSeek-Coder-6.7B [75], StarCoderBase-1B [76], StarCoderBase-3B [76], StarCoderBase-7B [76], and CodeLlama-7B [37]) for our experiments. These models feature a range of transformer architectures, including encoder-only, encoder-decoder, and decoder-only modes. All the models we selected have demonstrated encouraging results across various code understanding tasks like code clone detection, defect detection, etc. [30, 77–86]. Table 4 presents the details of the 10 base models, including their parameter sizes and architecture.

Table 4. Selected LMs and their corresponding parameter size and architecture

Model		#Params	Architecture
Small LMs	CodeBERT	125M	Encoder-only
	GraphCodeBERT	125M	
	UnixCoder	126M	
	CodeT5	220M	Encoder-decoder
LLMs	DeepSeek-Coder-1.3B	1.3B	Decoder-only
	DeepSeek-Coder-6.7B	6.7B	
	StarCoderBase-1B	1.1B	
	StarCoderBase-3B	3B	
	StarCoderBase-7B	7B	
	CodeLlama-7B	7B	

3.5 Metrics

We use metrics that align with previous research [19, 65, 87] to maintain consistency in evaluating the effectiveness of PEFT methods for code smell detection on LLMs. Specifically, we utilize precision, recall, F1-score, and MCC (Matthews Correlation Coefficient) for assessing the performance of detecting *CC* and *CM*. We include MCC because, unlike other metrics like F1-score, accounts for true negative instances [88]. These metrics are calculated using the following formulas:

$$\text{Precision} = \frac{TP}{TP + FP} \tag{3}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{4}$$

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5}$$

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \tag{6}$$

where TP, TN, FP, FN represent true positives, true negatives, false positives, and false negatives, respectively.

3.6 Implementation Details

The code for our experiments is implemented by the PEFT library[2] from Hugging Face. After reviewing prior studies [28–30, 46, 47] and observing the results of pilot experiments, we set the learning rate to $3e^{-4}$ for all PEFT methods and $1e^{-5}$ for full fine-tuning the small LMs. Throughout all the fine-tuning experiments, we utilized the AdamW optimizer [89] and fixed the Transformer input window size at 1,024. All the models were trained for 10 epochs, and we selected the epochs at which models exhibit minimum loss on the validation sets for inference. We executed all models on a single Linux GPU server equipped with a 48G NVIDIA RTX 6000 Ada Generation GPU. Additionally, we have released our datasets and source code online for replication purpose [1] where more details, including hyper-parameter values, can be found.

4 RESULTS

In this section, we first illustrate the experiment setup for each RQ formulated in Section 3, and we then present the results and findings for the four RQs.

4.1 RQ1: What is the effectiveness of different PEFT methods and full fine-tuning on various small LMs and LLMs?

Since our objective is to demonstrate the effectiveness of PEFT methods on both small and large LMs for code smell detection, we conducted a series of experiments involving four PEFT methods (i.e., prompt tuning, prefix tuning, LoRA, and $(IA)^3$) and a group of models with different parameter sizes. Each PEFT method was employed to fine-tune the 9 base models (see Section 3.4). Note that we only applied LoRA and $(IA)^3$ to Code-T5 as the current PEFT library does not support fine-tuning encoder-decoder models with prompt tuning and prefix tuning. We full fine-tuned all the small models due to resource constraints, as the computational demands of tuning all parameters in LLMs would surpass the available 48GB of GPU memory. Additionally, we reported and analyzed the computing resource consumption of these fine-tuning methods, and the results are shown in Figures 4, 5, and 6. Table 5 and Table 6 present the comprehensive comparison of small LMs and LLMs tuning with various PEFT techniques for detecting *CC* and *CM*, respectively.

4.1.1 Small LMs. Generally, GraphCodeBERT is the most effective in identifying *CC*, UnixCoder excels at detecting *CM*, and CodeT5 with PEFT methods performs the worst in classifying both types of method-level code smells. For *CC* and *CM*, CodeBERT demonstrates comparable results to GraphCodeBERT. For example, the F1-score and MCC for CodeBERT with prompt tuning in detecting *CM* are 85.20% and 72.21%, respectively, while the F1-score and MCC for GraphCodeBERT with prompt tuning for *CM* are 85.84% and 72.52%. These results are consistent with previous studies [79, 90, 91] that CodeBERT and GraphCodeBERT are powerful in code semantics learning, showing outstanding performance in Java code clone detection. GraphCodeBERT is built upon CodeBERT and additionally considers the inherent structure of code (i.e., data flow) in the pre-training stage [73]. Therefore, it is reasonable that GraphCodeBERT manifests a bit stronger capability than CodeBERT in code smell detection. UnixCoder also shows similar performance to GraphCodeBERT for *CC*, while it stands out as the best small model for *CM*, achieving an F1-score of 88.20% and an MCC of 77.56% with LoRA. For all the small LMs, CodeT5 with PEFT methods significantly

[2]https://huggingface.co/docs/peft/index

Table 5. Results of small LMs and LLMs with different PEFT methods and full fine-tuning on detecting *CC* (**bold**: highest values per model, **underline**: highest values among all models)

	Model	Method	Precision	Recall	F1	MCC
Small LMs	CodeBERT	Full FT	90.68%	50.00%	44.86%	0.00%
		Prompt Tuning	72.81%	69.55%	70.92%	42.23%
		Prefix Tuning	75.82%	73.23%	**74.38%**	**48.98%**
		LoRA	80.57%	69.03%	72.49%	48.24%
		$(IA)^3$	81.22%	68.86%	72.45%	48.53%
	GraphCodeBERT	Full FT	70.45%	75.65%	72.25%	45.80%
		Prompt Tuning	73.88%	71.85%	72.77%	45.69%
		Prefix Tuning	81.05%	73.00%	75.95%	53.44%
		LoRA	84.72%	69.67%	73.82%	52.27%
		$(IA)^3$	83.33%	74.30%	**<u>77.58%</u>**	**<u>56.91%</u>**
	UnixCoder	Full FT	67.67%	60.01%	61.69%	26.60%
		Prompt Tuning	73.93%	70.64%	72.03%	44.44%
		Prefix Tuning	78.73%	73.53%	**75.64%**	**52.00%**
		LoRA	82.12%	69.00%	72.74%	49.40%
		$(IA)^3$	74.33%	69.26%	71.21%	43.29%
	CodeT5	Full FT	76.94%	78.09%	**77.49%**	**55.02%**
		LoRA	63.08%	67.65%	46.82%	30.38%
		$(IA)^3$	62.42%	67.82%	63.11%	29.76%
LLMs	DeepSeek-Coder-1.3B	Prompt Tuning	75.25%	70.27%	72.23%	45.25%
		Prefix Tuning	75.07%	75.68%	**75.37%**	**50.75%**
		LoRA	78.35%	70.51%	73.28%	48.22%
		$(IA)^3$	75.74%	70.66%	72.65%	46.11%
	DeepSeek-Coder-6.7B	Prompt Tuning	74.96%	68.91%	71.13%	43.46%
		Prefix Tuning	79.36%	70.72%	**73.70%**	**49.33%**
		LoRA	75.55%	71.63%	73.26%	47.02%
		$(IA)^3$	75.79%	67.29%	69.99%	42.23%
	StarCoderBase-1B	Prompt Tuning	78.44%	71.06%	73.74%	48.95%
		Prefix Tuning	76.84%	70.95%	73.20%	47.42%
		LoRA	83.53%	54.82%	54.21%	25.41%
		$(IA)^3$	80.02%	70.31%	**73.51%**	**49.39%**
	StarCoderBase-3B	Prompt Tuning	73.67%	71.78%	**72.64%**	45.41%
		Prefix Tuning	74.18%	70.47%	72.01%	44.49%
		LoRA	69.39%	54.89%	54.72%	19.49%
		$(IA)^3$	79.58%	69.12%	72.39%	**47.57%**
	StarCoderBase-7B	Prompt Tuning	73.31%	68.97%	70.69%	42.05%
		Prefix Tuning	74.73%	72.62%	73.58%	47.30%
		LoRA	69.67%	65.46%	67.03%	34.88%
		$(IA)^3$	75.27%	76.48%	**75.84%**	**51.73%**
	CodeLlama-7B	Prompt Tuning	74.10%	71.92%	**72.90%**	45.97%
		Prefix Tuning	71.88%	71.13%	71.49%	43.00%
		LoRA	76.57%	73.20%	74.66%	**49.66%**
		$(IA)^3$	71.26%	71.57%	71.41%	42.83%

Table 6. Results of small LMs and LLMs with different PEFT methods and full fine-tuning on detecting *CM* (**bold**: highest values per model, **underline**: highest values among all models)

	Model	Method	Precision	Recall	F1	MCC
Small LMs	CodeBERT	Full FT	78.55%	53.84%	48.59%	20.95%
		Prompt Tuning	84.14%	88.18%	85.20%	72.21%
		Prefix Tuning	85.98%	88.42%	**86.95%**	**74.36%**
		LoRA	85.07%	86.07%	85.53%	71.13%
		$(IA)^3$	86.17%	87.32%	86.70%	73.48%
	GraphCodeBERT	Full FT	78.34%	82.63%	77.01%	60.81%
		Prompt Tuning	84.75%	87.83%	85.84%	**72.52%**
		Prefix Tuning	85.54%	86.79%	**86.11%**	72.33%
		LoRA	84.92%	86.97%	85.77%	71.86%
		$(IA)^3$	84.15%	86.40%	85.05%	70.51%
	UnixCoder	Full FT	76.24%	74.36%	75.13%	50.57%
		Prompt Tuning	84.64%	87.26%	85.64%	71.85%
		Prefix Tuning	86.89%	88.79%	87.70%	75.66%
		LoRA	86.99%	90.66%	**88.20%**	**77.56%**
		$(IA)^3$	86.26%	87.82%	86.95%	74.07%
	CodeT5	Full FT	85.66%	89.44%	**86.83%**	**75.00%**
		LoRA	78.08%	52.26%	45.44%	15.92%
		$(IA)^3$	42.00%	43.21%	42.36%	-14.73%
LLMs	DeepSeek-Coder-1.3B	Prompt Tuning	85.83%	88.05%	86.74%	73.85%
		Prefix Tuning	86.40%	89.81%	**87.57%**	**76.13%**
		LoRA	85.14%	87.79%	86.15%	72.88%
		$(IA)^3$	85.23%	87.59%	86.17%	72.78%
	DeepSeek-Coder-6.7B	Prompt Tuning	86.22%	88.92%	**87.26%**	75.08%
		Prefix Tuning	85.73%	89.96%	86.86%	**75.57%**
		LoRA	84.98%	87.48%	85.95%	72.42%
		$(IA)^3$	85.58%	83.14%	84.18%	68.68%
	StarCoderBase-1B	Prompt Tuning	85.68%	87.74%	86.54%	73.39%
		Prefix Tuning	84.82%	87.05%	85.72%	71.83%
		LoRA	84.81%	87.57%	85.84%	72.33%
		$(IA)^3$	86.30%	89.30%	**87.40%**	**75.53%**
	StarCoderBase-3B	Prompt Tuning	84.53%	86.24%	85.26%	70.74%
		Prefix Tuning	85.10%	86.38%	85.68%	71.48%
		LoRA	84.83%	89.44%	85.77%	74.13%
		$(IA)^3$	86.46%	88.38%	**87.28%**	**74.81%**
	StarCoderBase-7B	Prompt Tuning	83.64%	88.07%	84.53%	71.57%
		Prefix Tuning	85.35%	87.26%	86.16%	72.59%
		LoRA	85.64%	88.45%	86.70%	74.04%
		$(IA)^3$	85.72%	88.38%	**86.75%**	**74.05%**
	CodeLlama-7B	Prompt Tuning	74.05%	87.68%	86.00%	72.60%
		Prefix Tuning	84.10%	84.36%	84.23%	68.46%
		LoRA	85.39%	88.29%	**86.46%**	**73.62%**
		$(IA)^3$	84.07%	88.40%	85.05%	72.34%

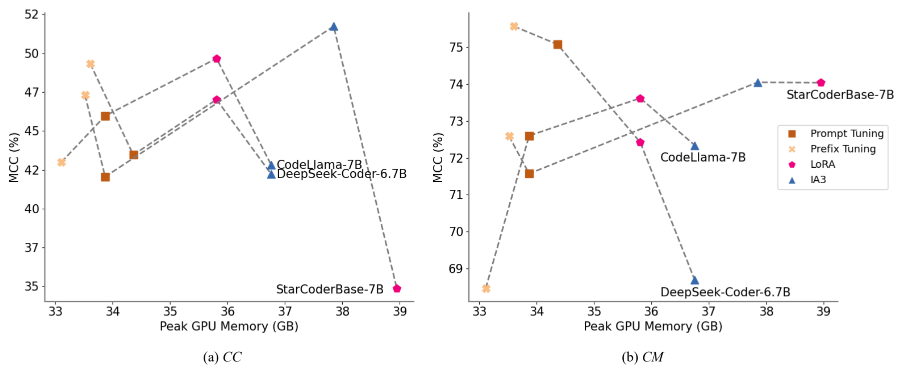


Fig. 4. Peak GPU memory of DeepSeek-Coder-6.7B, StarCoderBase-7B, and CodeLlama-7B with different PEFT methods on detecting *CC* and *CM*

lags behind the other three models with PEFT methods for both *CC* and *CM*, with MCC values well below 50%, and even a negative value when tuned with $(IA)^3$ for *CM*. This suggests that CodeT5 with PEFT methods struggles to perform effectively in code smell detection compared to the other models, highlighting the limitations of its PEFT adaptation. In contrast, fully fine-tuned CodeT5 outperforms all other fully fine-tuned small models in detecting both *CC* and *CM*. The stark performance gap between CodeT5 with full fine-tuning and PEFT methods could stem from CodeT5's architecture, which may depend more on full fine-tuning for optimal adaptation to the method-level code smell detection task. Compared to full fine-tuning, PEFT methods like LoRA and $(IA)^3$ might not provide sufficient adjustments to effectively adapt CodeT5's encoder-decoder architecture.

4.1.2 LLMs. DeepSeek-Coder-1.3B stands out as the top-performing LLM, consistently achieving the highest or on-par F1-scores and MCC values across all fine-tuning methods when compared to other LLMs. While StarCoderBase-7B with $(IA)^3$ gains higher F1-score and MCC value than DeepSeek-Coder-1.3B in detecting *CC*, it significantly underperforms when fine-tuned with LoRA. Specifically, the F1-score and MCC for DeepSeek-Coder-1.3B with LoRA are 73.28% and 48.22%, respectively, compared to 67.03% and 34.88% for StarCoderBase-7B. Notably, when fine-tuned with PEFT techniques, models with larger parameter sizes within the same family do not surpass, and in some cases, even slightly underperform their smaller counterparts. For example, when detecting *CC*, DeepSeekCoder-1.3B gains higher F1-scores and MCC values than DeepSeek-Coder-6.7B across the four PEFT methods, and StarCoderBase-1B with prompt tuning and prefix tuning also outperforms StarCoderBase-7B. This observation aligns with Fan *et al.*'s research [92], which shows that larger models do not always have better performance on code-related tasks.

Figure 4 illustrates the peak GPU memory consumption for the three largest models — DeepSeek-Coder-6.7B, StarCoderBase-7B, and CodeLlama-7B — using different PEFT techniques for code smell detection. It is evident that, for both CodeLlama-7B and DeepSeek-Coder-7B, $(IA)^3$ requires the highest computational resources, with LoRA ranking second. This observation is partly consistent with Li *et al.*'s work [28] that $(IA)^3$ tends to exhibit the highest memory usage. However, Li *et al.* [28] also found that LoRA consumes less memory than prefix tuning, which is not the case in our results. Unlike DeepSeek-Coder-6.7B and CodeLlama-7B, StarCoderBase-7B shows that LoRA demands more GPU memory than the other PEFT techniques.

4.1.3 Small LMs vs. LLMs. From Table 5 and Table 6, it can be observed that the best-performing models for detecting *CC* and *CM* are GraphCodeBERT with $(IA)^3$ and UnixCoder with LoRA, both of which are small LMs. More specifically, small LMs with PEFT techniques, with the exception of CodeT5, generally outperform or achieve comparable performance to LLMs. While Weyssow *et al.*'s work [47] highlights that LLMs significantly overtake small LMs in code generation tasks, our findings suggest that, for the binary classification task which identifies method-level code smells, small LMs with fewer number of trainable parameters excel beyond LLMs. This discrepancy might arise because the large capacity of LLMs introduces unnecessary complexity, which can obscure the essential patterns needed for the code smell detection task. In contrast, small LMs appear to possess an optimal capacity for capturing these crucial patterns without being encumbered by excessive complexity [93]. Another possible explanation for the discrepancy is that small LMs, being encoder-only models, are better suited to classification tasks like code smell detection. Encoder-only models are typically more efficient at tasks that require pattern recognition, such as binary classification, while LLMs are often optimized for generative tasks. This indicates that the advantages of LLMs observed in code generation tasks do not necessarily translate to all types of software engineering tasks, and larger models are not always superior when fine-tuned for different downstream applications.

4.1.4 Best PEFT Method. Overall, for both small LMs and LLMs in the DeepSeek-Coder family, prefix tuning emerges as the most effective technique for method-level code smell detection among the four PEFT methods investigated in this study. In contrast, for the StarCoderBase family, $(IA)^3$ tends to be the most effective technique. Meanwhile, for CodeLlama-7B, LoRA proves to be the best PEFT method. For small models, except for CodeT5, as well as LLMs like DeepSeek-Coder-1.3B and DeepSeek-Coder-6.7B, prefix tuning consistently yields the highest F1-scores and MCC values when detecting *CC* and *CM*, demonstrating its generally strong effectiveness in fine-tuning these models for code smell detection. In the StarCoderBase family, LLMs fine-tuned with $(IA)^3$ outperform those fine-tuned with other PEFT techniques. For example, when detecting *CM*, StarCoderBase-1B, StarCoderBase-3B, and StarCoderBase-7B all achieve their highest F1-scores and MCC values using $(IA)^3$. However, this does not apply to CodeLlama-7B, where using $(IA)^3$ results in the worst performance for detecting *CC* and the second-worst performance for detecting *CM*. On the other hand, CodeLlama-7B always achieves its best results on this task with LoRA. These findings contrast with those of previous studies [28, 47], which suggested that a single PEFT method could be the best across all models for a specific downstream task. For method-level code smell detection, the optimal PEFT method varies depending on the model. Notably, prompt tuning is the least effective PEFT method for adapting LMs to detect *CC* and *CM*, as most models tuned with prefix tuning show worse performance compared to those tuned with other PEFT techniques.

4.1.5 PEFT methods vs. Full Fine-tuning. According to Table 5 and Table 6, we can find that all the four PEFT methods achieve better or at least comparable effectiveness than full fine-tuning on most small models for method-level code smell detection. For small LMs, excluding CodeT5, the F1-scores and MCC values of prompt tuning, prefix tuning, LoRA, and $(IA)^3$ all show varying degrees of improvement over full fine-tuning. Our results align with Liu *et al.*'s prior study [29], confirming that PEFT methods can match or surpass the performance of standard full fine-tuning for code understanding tasks while adjusting fewer number of trainable parameters. One possible reason for this could be that full fine-tuning adjusts all of the model's parameters, which may lead to overfitting, especially when the training data is limited. In contrast, PEFT methods, by focusing on specific parts of the model, allow for more efficient learning without the risk of overfitting to the task at hand.

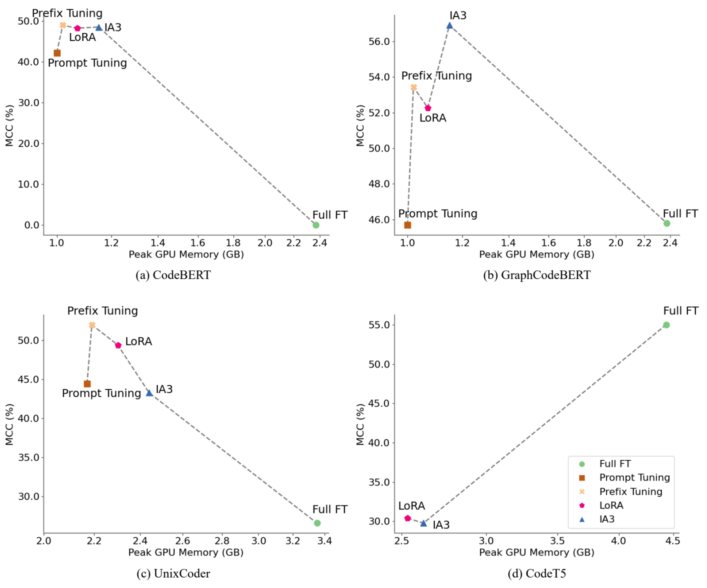


Fig. 5. Peak GPU memory of small LMs with different PEFT methods and full fine-tuning on detecting *CC*

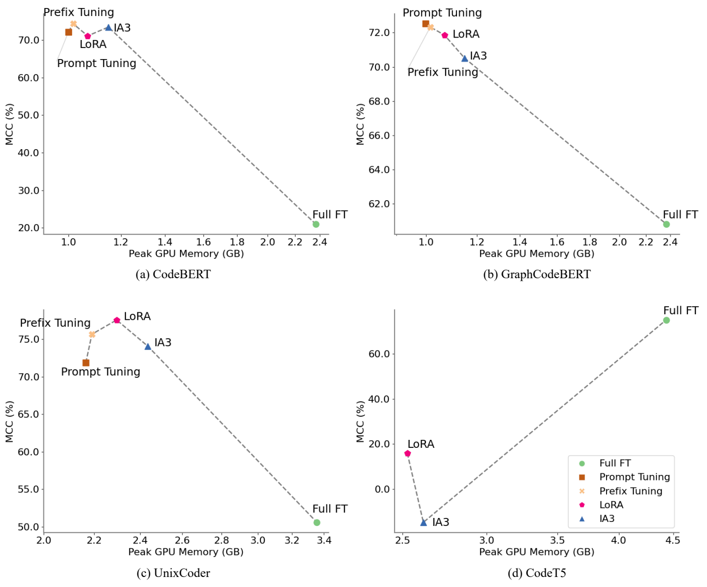


Fig. 6. Peak GPU memory of small LMs with different PEFT methods and full fine-tuning on detecting *CM*

PEFT methods focus on enhancing the effectiveness of LMs to levels similar to full fine-tuning, while aiming to minimize the use of computational resources, such as peak GPU memory consumption. Figure 5 and Figure 6 depict the peak GPU memory usage and corresponding MCC values of fine-tuning the four small models. It can be observed that prefix tuning and prompt tuning require significantly less GPU memory than other fine-tuning methods, yet they can still achieve considerable effectiveness. Notably, prefix tuning attains the highest or second-highest MCC value while using minimal GPU memory, proving that prefix tuning is the most efficient PEFT method for small models on code smell detection. When juxtaposed with full fine-tuning, all of the four PEFT methods lead to a substantial reduction in peak GPU memory usage while achieving better or similar results. This further validates the efficiency of PEFT methods for code smell detection relative to full fine-tuning.

Answer to RQ1: Small LMs, with the exception of CodeT5, generally achieve better effectiveness than LLMs when using PEFT methods for code smell detection. The most effective PEFT method varies across different models. All PEFT methods outperform or match the performance of full fine-tuning while requiring less peak GPU memory, highlighting the efficiency of PEFT techniques compared to full fine-tuning for detecting *CC* and *CM*.

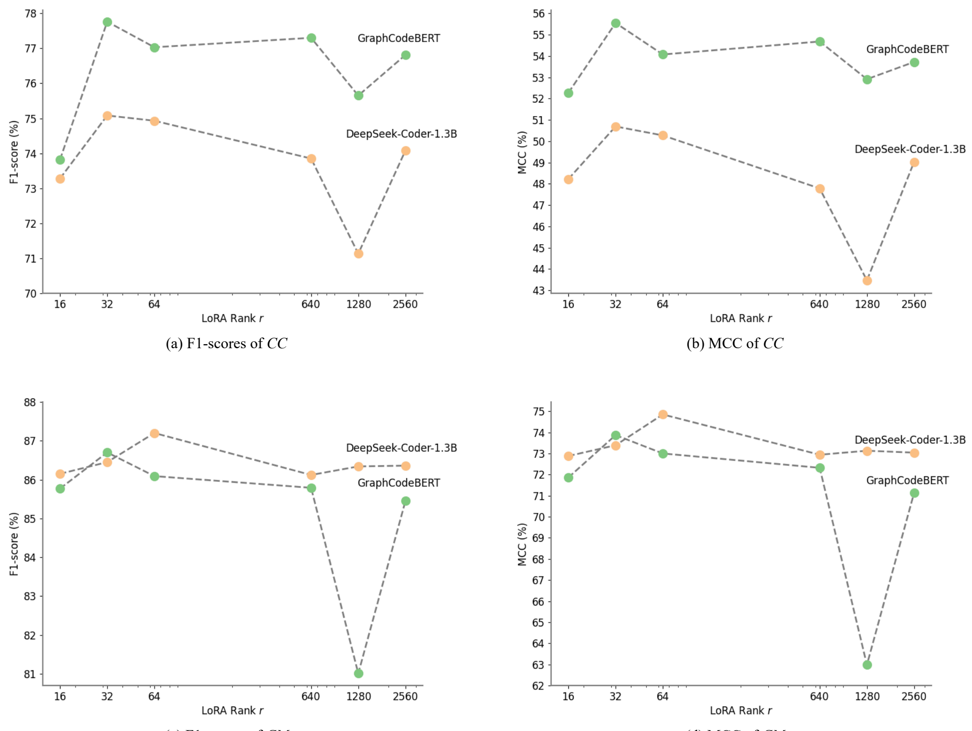
4.2 RQ2: What is the effectiveness of PEFT methods with different hyper-parameters?

Given the restrictions of our computational resources and popularity of LoRA, we focused on a single PEFT method, LoRA, to investigate how small LMs and LLMs perform under different hyper-parameter settings. Specifically, we used LoRA to tune a small LM (GraphCodeBERT) and an LLM (DeepSeek-Coder-1.3B), adapting the value of LoRA rank r in the range of {16, 32, 64, 640, 1,280, 2,560}. At the same time, we kept other hyper-parameters of LoRA constant, with the LoRA alpha set to 16 and the LoRA dropout set to 0.1. The experiment results are presented in Figure 7.

The results in Figure 7 show that both CodeBERT and DeepSeek-Coder-1.3B achieve their best performance in detecting *CC* when fine-tuned with LoRA at an r value of 32. For *CM*, GraphCodeBERT performs best at a LoRA r value of 32, while DeepSeek-Coder-1.3B reaches its peak performance at $r = 64$. As the r value increases from 16 to 2,560, the effectiveness of code smell detection initially increases, then decreases, and finally rises again at $r = 2,560$, but it does not exceed the level at $r = 16$.

This pattern can be explained by the increase in trainable parameters as r rises, which initially enhances the performance by allowing the models to learn more complex data patterns. However, beyond a certain threshold, the challenge of tuning these additional parameters effectively, especially with relatively limited amount of data, leads to diminishing returns. Without pre-training for these extra parameters, the models are prone to overfitting, where they may learn noise or irrelevant patterns, limiting their generalization capability. Consequently, as the number of trainable parameters grows beyond what the available training data can support, the models' performance may plateau or even decline. At $r = 2,560$, the models appear to utilize the expanded parameter space somewhat more effectively, leading to a slight performance recovery. However, this does not surpass the performance at $r = 16$, suggesting that the added parameters introduce more complexity than the data can justify. These results highlight that increasing the number of trainable parameters does not always lead to linear performance gains.

Answer to RQ2: Increasing the number of trainable parameters by adjusting hyper-parameters does not necessarily ensure effectiveness enhancement for code smell detection task. In fact, the effectiveness often declines as the number of trainable parameters continues to increase.



(a) F1-scores of *CC* (b) MCC of *CC*

(c) F1-scores of *CM* (d) MCC of *CM*

Fig. 7. F1-scores and MCC of GraphCodeBERT and DeepSeek-Coder-1.3B using LoRA on detecting *CC* and *CM* with different LoRA rank values

4.3 RQ3: What is the effectiveness of PEFT methods in low-resource scenarios?

To simulate the limited availability of data in low-resource scenarios, following previous works [29, 30], we randomly selected 100, 200, 500, and 1,000 samples from our datasets while maintaining the proportions of the original data distribution. In this RQ, we employed the four PEFT methods and full fine-tuning on a small LM (GraphCodeBERT) and an LLM (DeepSeek-Coder-1.3B) to conduct the experiments. The experiment results are shown in Table 7.

According to the results reported in Table 7, it is evident that when the data is limited to 100 or 200 samples, the effectiveness of both GraphCodeBERT and DeepSeek-Coder-1.3B with PEFT methods in detecting method-level code smells drops dramatically. This significant decline in performance highlights the challenges small and large LMs face in low-resource scenarios. However, when the number of training samples of *CC* or *CM* increases to 1,000, both models show improved performance compared to when only 100 samples are available. This improvement underscores that a larger dataset enables both small and large LMs using PEFT methods to achieve substantially better performance in code smell detection. Moreover, with 1,000 samples, GraphCodeBERT and DeepSeekCoder-1.3B with PEFT methods are able to yield reasonably good results compared to training with the original datasets for both types of method-level code smells. For example, the F1-scores of GraphCodeBERT training with 1,000 samples only degrade by no more than 3.26%, and the MCC decrease by no more than 7.22% compared to training with sufficient data for detecting *CM*,

Table 7. Results of GrahphCodeBERT and DeepSeek-Coder-1.3B with different PEFT methods and full fine-tuning on detecting *CC* and *CM* in low-resource scenarios (**bold**: highest values per scenario for a model, <u>**underline**</u>: highest values among all scenarios for a model)

Model	#Training Samples	Method	*CC*		*CM*	
			F1	MCC	F1	MCC
GraphCodeBERT	100	Full FT	44.86%	0.00%	**83.87%**	**69.52%**
		Prompt Tuning	47.18%	6.84%	80.25%	61.85%
		Prefix Tuning	63.94%	32.85%	76.33%	52.67%
		LoRA	57.34%	25.28%	82.44%	64.96%
		$(IA)^3$	**69.63%**	**39.47%**	81.32%	65.84%
	200	Full FT	52.48%	21.18%	**86.26%**	**73.93%**
		Prompt Tuning	67.05%	34.12%	83.26%	67.26%
		Prefix Tuning	**72.47%**	**46.00%**	81.15%	62.49%
		LoRA	66.61%	41.44%	85.20%	71.60%
		$(IA)^3$	71.92%	44.12%	84.05%	68.31%
	500	Full FT	67.78%	43.34%	<u>**86.69%**</u>	<u>**74.50%**</u>
		Prompt Tuning	62.44%	27.96%	80.20%	60.40%
		Prefix Tuning	69.92%	41.74%	83.26%	66.72%
		LoRA	**71.08%**	**45.30%**	84.92%	69.84%
		$(IA)^3$	67.87%	39.54%	84.25%	69.17%
	1,000	Full FT	<u>**75.22%**</u>	<u>**50.81%**</u>	85.49%	73.83%
		Prompt Tuning	69.63%	39.74%	82.58%	65.30%
		Prefix Tuning	73.44%	47.02%	84.16%	68.84%
		LoRA	74.76%	49.54%	**86.67%**	**74.16%**
		$(IA)^3$	63.95%	35.41%	84.07%	68.13%
DeepSeekCoder-1.3B	100	Full FT	44.86%	0.00%	81.48%	**68.21%**
		Prompt Tuning	47.43%	14.34%	78.45%	56.93%
		Prefix Tuning	**59.82%**	**20.70%**	82.35%	67.39%
		LoRA	54.90%	16.98%	**83.48%**	67.47%
		$(IA)^3$	50.93%	8.63%	74.02%	48.52%
	200	Full FT	**72.76%**	**50.97%**	**85.25%**	**72.17%**
		Prompt Tuning	60.80%	30.66%	77.63%	55.49%
		Prefix Tuning	69.20%	39.65%	82.52%	67.05%
		LoRA	66.65%	36.72%	83.18%	66.49%
		$(IA)^3$	64.97%	32.56%	77.91%	55.91%
	500	Full FT	66.34%	40.34%	83.32%	69.66%
		Prompt Tuning	62.13%	30.85%	80.12%	60.25%
		Prefix Tuning	71.96%	44.14%	**83.91%**	68.70%
		LoRA	**72.94%**	**45.89%**	83.79%	**69.70%**
		$(IA)^3$	63.07%	34.91%	79.17%	58.36%
	1,000	Full FT	72.80%	45.63%	82.50%	65.25%
		Prompt Tuning	70.24%	42.31%	82.04%	64.10%
		Prefix Tuning	70.26%	42.97%	84.43%	69.58%
		LoRA	71.84%	44.27%	<u>**86.13%**</u>	<u>**73.00%**</u>
		$(IA)^3$	<u>**73.33%**</u>	<u>**46.68%**</u>	80.33%	60.65%

despite the dataset size being reduced by 90.35% ((10,366 - 1,000) / 10,366). This highlights the benefit of PEFT in resource-limited settings, as it enables the reduction of not only training parameters but also the amount of required training data, while still maintaining strong effectiveness on code smell detection. Notably, when the number of training samples increases to 1,000, GraphCodeBERT and DeepSeek-Coder-1.3B with PEFT methods demonstrate effectiveness in detecting *CC* and *CM* that is on par with, or even exceeds, that of full fine-tuning, further highlighting the superiority of PEFT methods in low-resource scenarios.

Secondly, compared to the results in Table 5 and Table 6, we can find that GraphCodeBERT with full fine-tuning performs better in low-resource scenarios than when there is sufficient data. This improvement can be attributed to the fact that, with ample data, full fine-tuning may lead to overfitting, causing the model to memorize rather than generalize the data. In contrast, in low-resource scenarios, the model is forced to learn more generalizable patterns, thus improving its performance.

Finally, in low-resource scenarios, LoRA tends to be the most effective PEFT method. As indicated by the results of RQ1 (see Section 4.1.4), for both GraphCodeBERT and DeepSeek-Coder-1.3B, fine-tuning with prefix tuning enables them to achieve their best performance compared to the other three PEFT methods. However, when data is scarce, GraphCodeBERT and DeepSeek-Coder-1.3B often attain their highest F1-scores or MCC values through LoRA. It suggests that under different resource settings, the most effective PEFT method may vary.

Answer to RQ3: In low-resource scenarios, the effectiveness of PEFT methods for code smell detection diminishes considerably when data is limited to 100 or 200 samples. However, as the number of training samples increases, the performance of PEFT methods shows significant improvement, and in some cases, even surpasses that of full fine-tuning.

4.4 RQ4: What is the effectiveness of PEFT methods compared to state-of-the-art code smell detection approaches?

Prompt Template for Detecting Complex Conditional with GPT-3.5-turbo

You are a professional Java programmer. Given a piece of Java code, determine whether it contains “Complex Conditional” code smell. If it does, return only “Yes”, else return “No”. Definition of “Complex Conditional”: This code smell occurs when a conditional statement is excessively intricate.

Input: [*Code Sample*]
Output: [*Yes/No*]

Prompt Template for Detecting Complex Method with GPT-3.5-turbo

You are a professional Java programmer. Given a piece of Java code, determine whether it contains “Complex Method” code smell. If it does, return only “Yes”, else return “No”. Definition of “Complex Method”: This code smell occurs when a method exhibits high cyclomatic complexity.

Input: [*Code Sample*]
Output: [*Yes/No*]

Traditional heuristic-based tools for code smell detection have been widely used in academia [56, 67, 68], while DL has recently gained widespread adoption in research on code smell detection, achieving impressive performance [66]. In this RQ, we established both heuristic-based (i.e., DesigniteJava) and best-in-class DL detectors (i.e., DeepSmells [65] and AE-Dense [19]), as well as a general-purpose LLM (i.e., GPT-3.5-turobo[3]) which has not been specifically tuned for method-level code smell detection, as baselines. To obtain valid responses from GPT-3.5-turbo in a zero-shot setting [94], we designed the above prompt templates that include both instructions and definitions of *CC* or *CM* to guide ChatGPT in annotating the test sets. GraphCodeBERT with $(IA)^3$ for *CC* and UnixCoder with LoRA for *CM* are selected for comparison. The results of this evaluation are shown in Table 8.

From the results reported in Table 8, we can find that GraphCodeBERT with $(IA)^3$ and UnixCoder with LoRA exhibit significantly better performance than the state-of-the-art code smell detection approaches and GPT-3.5-turbo in detecting *CC* and *CM*, respectively. Specifically, for *CC*, GraphCodeBERT with $(IA)^3$ significantly surpasses the effectiveness of the other four baseline methods with an F1-score disparity ranging from 23.41% to 47.94% and an MCC difference ranging from 13.24% to 50.59%. For *CM*, UnixCoder with LoRA achieves an F1-score improvement ranging from 11.22% to 54.32% and an MCC improvement ranging from 10.28% to 72.91%.

Table 8. Results of GraphCodeBERT with $(IA)^3$ and UnixCoder with LoRA compared to state-of-the-art code smell detection approaches on detecting *CC* and *CM* (**bold**: highest values per code smell)

Code Smell	Method	Precision	Recall	F1	MCC
CC	DesigniteJava	34.58%	98.74%	51.22%	43.67%
	DeepSmells	21.61%	47.17%	29.64%	6.32%
	AE-Dense	42.90%	53.82%	47.74%	15.20%
	GPT-3.5-turbo	42.86%	73.58%	54.17%	42.66%
	GraphCodeBERT with $(IA)^3$	83.33%	74.30%	**77.58%**	**56.91%**
CM	DesigniteJava	61.88%	99.27%	76.23%	65.92%
	DeepSmells	33.82%	56.80%	42.39%	4.65%
	AE-Dense	24.33%	55.76%	33.88%	9.78%
	GPT-3.5-turbo	81.35%	73.06%	76.98%	67.28%
	UnixCoder with LoRA	86.99%	90.66%	**88.20%**	**77.56%**

Answer to RQ4: In comparison with existing code smell detection approaches, LMs tuned with PEFT methods exhibit notable performance improvements on identifying *CC* and *CM*, highlighting the effectiveness of PEFT methods for code smell detection.

5 DISCUSSION

Approaches with improved effectiveness for *CC* code smell detection are needed: According to Fowler [2], four factors contribute to making programs difficult to work with, one of which is *Complex Conditional* (*CC*) logic that complicates modification. The presence of *CC* in source code hinders testing and maintenance [2, 95, 96]. Thus, identifying *CC* code smells in software can significantly improve code quality [95]. However, the effectiveness of existing methods for detecting this type of code smell is considerably limited. As shown in Table 8, existing methods – whether heuristic-based (e.g., DesigniteJava), DL-based (e.g., DeepSmells and AE-Dense), or

[3]https://platform.openai.com/docs/models/o1

GPT-3.5-turbo — demonstrate limited success in detecting *CC*, with all MCC values below 45%, and some even as low as 6.32%. Although our proposed method, GraphCodeBERT with $(IA)^3$, improved the performance of *CC* code smell detection significantly, reaching an F1-score of 77.58% and an MCC value of 56.91%, it still falls short of expectations due to challenges such as the considerable existence of false positives and negatives, which may limit its practical applicability. Accurate identification of *CC* code smells can help developers better refactor the code, ultimately improving the maintainability of the software system. Despite the progress made, the detection of *CC* remains a challenging task that requires further improvements.

Model size is not the decisive factor contributing to the effectiveness of code smell detection: Previous research has demonstrated that larger models with more parameters tend to deliver better performance on code-related tasks [37, 47, 97]. For example, Weyssow *et al.* [47] suggested a potential shift in the software engineering domain, moving away from the long-established practice of fine-tuning smaller LMs like CodeT5 over the past 3-4 years, and towards embracing larger LLMs. However, the results of RQ1 indicate that, for the method-level code smell detection task, model size is not the sole determinant of performance. In fact, all the small models, with the exception of CodeT5, generally perform as well as or better than larger LLMs in detecting *CC* and *CM* (see Section 4.1.3). Furthermore, within the same model family, larger models do not always outperform its smaller-sized counterparts (see Section 4.1.2). In contrast, the alignment between the model and the specific task at hand seems to play a crucial role in determining effectiveness for code smell detection. While CodeT5, when fine-tuned with PEFT methods, has demonstrated strong performance in tasks such as code summarization and code change prediction [29, 92], it surprisingly underperforms in the task of code smell detection. This suggests that models may exhibit varying performance across different downstream tasks, emphasizing the importance of aligning a model's design and fine-tuning with the specific requirements of each target task. Therefore, it is essential to focus on selecting models that are well-suited to the method-level code smell detection task, rather than relying solely on model size as a performance indicator.

Considering the model used and the availability of resources when selecting the appropriate PEFT method for code smell detection: Based on the results of RQ1 and RQ3, we can find that for method-level code smell detection, the most effective PEFT method can differ significantly depending on the models and the amount of available data. From Table 5 and Table 6, it is obvious that each model responds to different PEFT techniques in unique ways, suggesting that the optimal PEFT method is rather model-dependent. The effectiveness of PEFT methods can change even for the same model under different resource scenarios. For instance, in low-resource settings with limited data, the best PEFT method for models like GraphCodeBERT and DeepSeek-Coder-1.3B shifts from prefix tuning to LoRA. Moreover, peak GPU consumption can vary across different models (see Section 4.1.2). Generally, $(IA)^3$ requires more GPU resources than LoRA for code smell detection, whereas the reverse is true for StarCoderBase-7B. These variations underscore that there is no universally best PEFT method for method-level code smell detection; instead, the most suitable method depends on the specific model and resource constraints. Therefore, we recommend that practitioners consider both the model's characteristics and the availability of resources, including data volume and GPU memory, when selecting the appropriate PEFT method for a given task.

Start model selection for code smell detection with a smaller dataset to optimize resource usage: In low-resource scenarios, when the dataset size was reduced by more than 90%, the models' effectiveness did not degrade significantly, and they still yielded promising results, as shown in the results of RQ3 (see Section 4.3). This finding indicates that, even with a significantly smaller dataset, both small and large LMs can maintain robust performance for the code smell detection task. Consequently, beginning fine-tuning experiments with a smaller-size dataset can be an effective strategy to optimize resource usage. This approach allows researchers and practitioners to initially evaluate a wide range of LMs in a shorter time and wither fewer computational resources, saving substantial effort. It enables them to identify the most suitable model for the specific task through an initial evaluation before committing to the more resource-intensive process of training on larger datasets. Additionally, it helps in assessing the trade-offs between resource consumption and performance, providing insight into the models' scalability. By leveraging smaller datasets for initial evaluation, one can make informed decisions about which models are likely to perform well with limited resources, and further fine-tune or scale up as necessary, without wasting valuable time or computational power.

6 THREATS TO VALIDITY

Construct Validity: One primary threat to the construct validity of this study is the construction of the datasets. We first used DesigniteJava [31] to detect potential positive and negative samples, then manually reviewed all the candidate data to validate the datasets, ensuring there were no false positives or negatives.

First, relying solely on DesigniteJava for initial sample detection may introduce bias. However, DesigniteJava is a well-established and widely adopted tool in Java code smell detection [56, 67, 98], providing a robust foundation for the identification of potential code smells. We also acknowledge that future studies could benefit from incorporating additional tools in the initial code smell detection process, which may help mitigate the threat to construct validity by providing more diverse perspectives on the data.

Second, the manual validation process, while aimed at ensuring dataset accuracy, could introduce some subjectivity due to its reliance on human judgment. To minimize this threat, we conducted separate pilot validations for *CC* and *CM* before formally annotating all the data samples. This pilot phase helped ensure that the two annotators used the same criteria for validation, thus improving consistency and reducing potential bias in the dataset labeling.

External Validity: In this study, we evaluated the effectiveness of state-of-the-art PEFT methods using 4 small LMs and 6 LLMs. However, due to limited computational resources, we did not include larger models with over 7B parameters, which may impact the generalizability of our findings. Additionally, we only employed four popular and typical PEFT techniques, i.e., prompt tuning, prefix tuning, LoRA, and $(IA)^3$. Other PEFT methods, such as adapter tuning [99], could be explored in our future work. Moreover, our study only applied PEFT methods to detect Java code smells. Expanding this research to include code smells in other programming languages, such as C# and Python, could provide further insights into the effectiveness and adaptability of PEFT methods across diverse language-specific code smell patterns.

Internal Validity: The settings of hyper-parameters for the four PEFT methods and full fine-tuning play a crucial role in the performance of the models. Variations in these settings could lead to different results, potentially affecting the reliability of our findings. To mitigate this threat, we referred to previous works and conducted pilot experiments to determine the appropriate hyper-parameter values.

Additionally, data leakage represents another potential threat to internal validity. First, the poor zero-shot performance of GPT-3.5-turbo in detecting *CC* and *CM* code smells suggests that the model was not exposed to the test data (see Section 4.4), as we would expect stronger zero-shot effectiveness if GPT-3.5-turbo had memorized the data. Second, the significant improvements in performance for both small and large LMs following the application of PEFT techniques further suggest that memorization is unlikely to be an issue. Such substantial performance gains would not occur if the models had already memorized the test sets. Furthermore, Cao *et al.* [100] highlight that code LMs do not consistently exhibit reduced performance on data extending beyond their training cut-off dates. This careful management of data, along with the observed performance improvements after fine-tuning, ensures that data leakage was not a major threat, thereby maintaining the internal validity of our findings.

7 RELATED WORK

7.1 Code Smell Detection

Existing approaches to code smell detection predominantly rely on heuristic-based methods, which utilize predefined rules to identify problematic code patterns [7, 31, 32, 53, 101, 102]. These methods aim to flag potential smells by analyzing the source code against established standards. For instance, PMD [32], a widely used static code analysis tool, supports 16 programming languages and applies rule sets to detect code smells. These rule sets can be tailored to suit specific project requirements or design standards, making the tool adaptable for different contexts. Similarly, Sharma [31] introduced DesigniteJava, an advanced tool capable of identifying code smells at multiple levels — ranging from architecture and design to implementation.

Traditional heuristic-based code smell detectors face inherent challenges, such as difficulty in defining effective thresholds for detection. In response, Machine Learning (ML) and Deep Learning (DL) techniques have gained traction as promising alternatives. Maiga *et al.* [103] implemented SVMDetect based on an ML technique — support vector machines. Fontana *et al.* [18] evaluated 16 ML algorithms across four types of code smells using 74 software systems, reporting that J48 and Random Forest delivered the highest performance.

Liu *et al.* [20] utilized deep neural networks and advanced DL techniques to perform useful feature selection from source code and map these features to labels (smelly or not). Sharma *et al.* [19] trained Convolution Neural Network (CNN), Recurrent Neural Network (RNN), and autoencoder DL models in different configurations to detect code smells. Their work confirms the feasibility of applying transfer-learning for code smell detection. Ho *et al.* [65] developed DeepSmells based on a CNN and long short-term memory (LSTM) networks for detecting code smells. They conducted an empirical study to compare DeepSmells with other well-established detection tools. Their results demonstrate that DeepSmells outperforms the baselines, achieving superior performance. They also anticipated that leveraging CodeBERT would enhance the effectiveness of code smell detection. Recently, Wu *et al.* [104] trained a Mixture of Experts (MoE) model, leveraging specialized expert toolsets for code smell detection. They subsequently utilized the detection results to prompt LLMs for refactoring.

Most prevailing ML- or DL-based approaches for code smell detection train models from scratch. Compared to their works, we harness the power of LLMs using PEFT methods to detect two types of complex method-level code smells (*Complex Conditional* and *Complex Method*), aiming to enhance the models' ability to capture the nuanced relationships between different types of method-level code smells.

7.2 PEFT Methods on Software Engineering Tasks

Several studies have evaluated PEFT methods on various SE tasks. Wang *et al.* [30] fine-tuned UniXcoder and CodeT5 by inserting the parameter-efficient structure adapter for code search and summarization. According to their results, adapter tuning has shown effectiveness in cross-lingual and low-resource scenarios, surpassing full fine-tuning and effectively mitigating catastrophic forgetting. Liu *et al.* [29] carried out experiments by fine-tuning code models with adapter, prefix tuning, LoRA, and MHM [105] on both the code understanding task and code generation task. Their findings indicate that adapter tuning and LoRA can achieve promising performance on these two SE tasks with reduced training costs and memory requirements. Liu *et al.* [46] examined whether adapter tuning and LoRA can perform well on code-change-related tasks, specially just-in-time defect prediction and commit message generation. Their study demonstrates that PEFT methods can serve as a powerful alternative to full fine-tuning for dynamic code comprehension. Weyssow *et al.* [47] compared PEFT methods, including prompt tuning and others, with In-Context Learning (ICL) in terms of code generation. Their investigation reveals the superiority of PEFT over ICL and highlights the potential opportunities of tuning LLMs in diverse software engineering scenarios. Li *et al.* [28] created an instruction dataset of Automated Program Repair (APR) using prompt engineering. They employed this dataset to fine-tune four LLMs using four different PEFT methods. The results show that $(IA)^3$ achieves the highest fixing ability compared to prompt tuning, prefix tuning, and LoRA.

These studies focus on evaluating PEFT methods across a range of software engineering tasks, such as code summarization and code generation. Compared to their works, our work pays specific attention to a binary classification task, aiming to assess the effectiveness of LLMs with PEFT methods in identifying different types of code smells.

8 CONCLUSIONS AND FUTURE WORK

In this work, we investigated the effectiveness of various PEFT methods on different LMs for code smell detection. We also evaluated how the performance of these PEFT methods varies under different conditions, such as varying hyper-parameter settings and the availability of training data. Our analysis reveals the efficiency of PEFT methods compared to full fine-tuning, underscoring the potential of PEFT methods to enhance method-level code smell detection and offer valuable insights for optimizing LMs for other software engineering tasks. Additionally, we have constructed high-quality datasets for *CC* and *CM* code smells, and we provided the replication package of this work online [1] to facilitate future research and ensure the reproducibility of our study results.

In the next step, we plan to further explore how PEFT performs in cross-language scenarios for code smell detection. Moreover, we plan to investigate the application of PEFT methods for fine-tuning LMs to assist in code smell refactoring, thereby improving overall code quality and maintainability by providing just-in-time code smell detection and refactoring suggestions.

DATA AVAILABILITY

The replication package for this work has been made available at https://github.com/MabelQi/PEFT4CSD.

ACKNOWLEDGMENTS

This work has been partially supported by the National Natural Science Foundation of China (NSFC) with Grant No. 62172311.

REFERENCES

[1] B. Zhang, P. Liang, X. Zhou, X. Zhou, D. Lo, Q. Feng, Z. Li, and L. Li, "Replication package for the paper: A comprehensive evaluation of parameter-efficient fine-tuning on method-level code smell detection," 2024. https://github.com/MabelQi/PEFT4CSD.
[2] M. Fowler, *Refactoring - Improving the Design of Existing Code.* Addison-Wesley, 1999.
[3] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.
[4] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns blob and spaghetti code on program comprehension," in *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 181–190, IEEE, 2011.

[5] A. Chatzigeorgiou and A. Manakos, “Investigating the evolution of bad smells in object-oriented code,” in *Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pp. 106–115, IEEE, 2010.

[6] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems.* Springer Science & Business Media, 2007.

[7] N. Moha, Y. Guéhéneuc, L. Duchien, and A. L. Meur, “DECOR: A method for the specification and detection of code and design smells,” *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20–36, 2010.

[8] S. Prokić, N. Luburić, J. Slivka, and A. Kovačević, “Prescriptive procedure for manual code smell annotation,” *Science of Computer Programming*, vol. 238, p. 103168, 2024.

[9] R. Marinescu, “Detection strategies: metrics-based rules for detecting design flaws,” in *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM)*, pp. 350–359, IEEE, 2004.

[10] R. Morales, Z. Soh, F. Khomh, G. Antoniol, and F. Chicano, “On the use of developers’ context for automatic refactoring of software anti-patterns,” *Journal of Systems and Software*, vol. 128, pp. 236–251, 2017.

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

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

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

[14] F. A. Fontana, J. Dietrich, B. Walter, A. Yamashita, and M. Zanoni, “Antipattern and code smell false positives: Preliminary conceptualization and classification,” in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 609–613, IEEE, 2016.

[15] M. V. Mäntylä and C. Lassenius, “Subjective evaluation of software evolvability using code smells: An empirical study,” *Empirical Software Engineering*, vol. 11, pp. 395–431, 2006.

[16] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, “Building empirical support for automated code smell detection,” in *Proceedings of the 4th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1–10, ACM, 2010.

[17] F. A. Fontana, M. Zanoni, A. Marino, and M. Mäntylä, “Code smell detection: Towards a machine learning-based approach,” in *Proceedings of 29th the IEEE International Conference on Software Maintenance (ICSM)*, pp. 396–399, IEEE, 2013.

[18] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, “Comparing and experimenting machine learning techniques for code smell detection,” *Empirical Software Engineering*, vol. 21, pp. 1143–1191, 2016.

[19] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, “Code smell detection by deep direct-learning and transfer-learning,” *Journal of Systems and Software*, vol. 176, p. 110936, 2021.

[20] H. Liu, J. Jin, Z. Xu, Y. Zou, Y. Bu, and L. Zhang, “Deep learning based code smell detection,” *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1811–1837, 2021.

[21] D. D. Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. D. Lucia, “Detecting code smells using machine learning techniques: Are we there yet?,” in *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 612–621, IEEE, 2018.

[22] J. Pereira dos Reis, F. Brito e Abreu, G. de Figueiredo Carneiro, and C. Anslow, “Code smells detection and visualization: a systematic literature review,” *Archives of Computational Methods in Engineering*, vol. 29, no. 1, pp. 47–94, 2022.

[23] A. Alazba, H. Aljamaan, and M. Alshayeb, “Deep learning approaches for bad smell detection: a systematic literature review,” *Empirical Software Engineering*, vol. 28, no. 3, p. 77, 2023.

[24] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, “CodeBERT: A pre-trained model for programming and natural languages,” in *Proceedings of the 28th International Conference on Empirical Methods in Natural Language Processing (EMNLP): Findings*, pp. 1536–1547, ACL, 2020.

[25] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” in *Proceedings of the 26th International Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 8696–8708, ACL, 2021.

[26] J. Lu, L. Yu, X. Li, L. Yang, and C. Zuo, “Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning,” in *Proceedings of the 34th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pp. 647–658, IEEE, 2023.

[27] T. K. Le, S. Alimadadi, and S. Y. Ko, “A study of vulnerability repair in javascript programs with large language models,” in *Proceedings of the 33rd ACM on Web Conference (WWW) Companion*, pp. 666–669, ACM, 2024.

[28] G. Li, C. Zhi, J. Chen, J. Han, and S. Deng, “A comprehensive evaluation of parameter-efficient fine-tuning on automated program repair,” *arXiv preprint arXiv:2406.05639*, 2024.

[29] J. Liu, C. Sha, and X. Peng, "An empirical study of parameter-efficient fine-tuning methods for pre-trained code models," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 397–408, IEEE, 2023.

[30] D. Wang, B. Chen, S. Li, W. Luo, S. Peng, W. Dong, and X. Liao, "One adapter for all programming languages? adapter tuning for code search and summarization," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 5–16, IEEE, 2023.

[31] T. Sharma, "Designitejava," 2018. https://github.com/tushartushar/DesigniteJava.

[32] "PMD," 2024. https://pmd.github.io/.

[33] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, and R. Wettel, "iplasma: An integrated platform for quality assessment of object-oriented design," in *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM): Industrial and Tool Volume*, pp. 77–80, IEEE, 2005.

[34] P. Ardimento, L. Aversano, M. L. Bernardi, M. Cimitile, and M. Iammarino, "Temporal convolutional networks for just-in-time design smells prediction using fine-grained software metrics," *Neurocomputing*, vol. 463, pp. 454–471, 2021.

[35] P. Ardimento, L. Aversano, M. L. Bernardi, M. Cimitile, and M. Iammarino, "Transfer learning for just-in-time design smells prediction using temporal convolutional networks," in *Proceedings of the 16th International Conference on Software Technologies (ICSOFT)*, pp. 310–317, SciTePress, 2021.

[36] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[37] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, and et al., "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2024.

[38] X. Qiu, T. Sun, Y. Xu, Y. Shao, N. Dai, and X. Huang, "Pre-trained models for natural language processing: A survey," *Science China Technological Sciences*, vol. 63, no. 10, pp. 1872–1897, 2020.

[39] L. Hu, Z. Liu, Z. Zhao, L. Hou, L. Nie, and J. Li, "A survey of knowledge enhanced pre-trained language models," *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 4, pp. 1413–1430, 2024.

[40] N. Ding, Y. Qin, G. Yang, F. Wei, Z. Yang, Y. Su, S. Hu, Y. Chen, C.-M. Chan, W. Chen, *et al.*, "Parameter-efficient fine-tuning of large-scale pre-trained language models," *Nature Machine Intelligence*, vol. 5, no. 3, pp. 220–235, 2023.

[41] Z. Han, C. Gao, J. Liu, S. Q. Zhang, *et al.*, "Parameter-efficient fine-tuning for large models: A comprehensive survey," *arXiv preprint arXiv:2403.14608*, 2024.

[42] B. Lester, R. Al-Rfou, and N. Constant, "The power of scale for parameter-efficient prompt tuning," in *Proceedings of the 26th International Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 3045–3059, ACL, 2021.

[43] X. L. Li and P. Liang, "Prefix-tuning: Optimizing continuous prompts for generation," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (ACL/IJCNLP)*, pp. 4582–4597, ACL, 2021.

[44] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," in *Proceedings of the 10th International Conference on Learning Representations (ICLR)*, pp. 1–13, OpenReview.net, 2022.

[45] H. Liu, D. Tam, M. Muqeeth, J. Mohta, T. Huang, M. Bansal, and C. Raffel, "Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning," in *Proceedings of the 36th Annual Conference on Neural Information Processing Systems (NeurIPS)*, vol. 35, pp. 1950–1965, Curran Associates, Inc., 2022.

[46] S. Liu, J. Keung, Z. Yang, F. Liu, Q. Zhou, and Y. Liao, "Delving into parameter-efficient fine-tuning in code change learning: An empirical study," in *Proceedings of the 31st IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 465–476, IEEE, 2024.

[47] M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui, "Exploring parameter-efficient fine-tuning techniques for code generation with large language models," *arXiv preprint arXiv:2308.10462*, 2023.

[48] D. Gupta, A. Bhatti, and S. Parmar, "Beyond lora: Exploring efficient fine-tuning techniques for time series foundational models," *arXiv preprint arXiv:2409.11302*, 2024.

[49] L. Zhang, J. Wu, D. Zhou, and G. Xu, "Star: Constraint lora with dynamic active learning for data-efficient fine-tuning of large language models," *arXiv preprint arXiv:2403.01165*, 2024.

[50] M. Zakeri-Nasrabadi, S. Parsa, E. Esmaili, and F. Palomba, "A systematic literature review on the code smells datasets and validation mechanisms," *ACM Computing Surveys*, vol. 55, no. 13s, pp. 1–48, 2023.

[51] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, 2015.

[52] F. Palomba, A. Panichella, A. D. Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC)*, pp. 1–10, IEEE, 2016.

[53] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, “Jdeodorant: Identification and removal of feature envy bad smells,” in *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM)*, pp. 519–520, IEEE, 2007.

[54] F. A. Fontana, P. Braione, and M. Zanoni, “Automatic detection of bad smells in code: An experimental assessment,” *Journal of Object Technology*, vol. 11, no. 2, pp. 5–1, 2012.

[55] F. Pecorelli, S. Lujan, V. Lenarduzzi, F. Palomba, and A. D. Lucia, “On the adequacy of static analysis warnings with respect to code smell prediction,” *Empirical Software Engineering*, vol. 27, no. 3, p. 64, 2022.

[56] T. Sharma and M. Kessentini, “Qscored: A large dataset of code smells and quality metrics,” in *Proceedings of the IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 590–594, IEEE, 2021.

[57] B. Nguyen Thanh, M. Nguyen N. H., H. Le Thi My, and B. Nguyen Thanh, “ml-codesmell: A code smell prediction dataset for machine learning approaches,” in *Proceedings of the 11th International Symposium on Information and Communication Technology (SoICT)*, pp. 368—-374, ACM, 2022.

[58] L. Madeyski and T. Lewowski, “Detecting code smells using industry-relevant data,” *Information and Software Technology*, vol. 155, p. 107112, 2023.

[59] L. Madeyski and T. Lewowski, “Mlcq: Industry-relevant code smell data set,” in *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pp. 342–347, ACM, 2020.

[60] F. Palomba, D. Di Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, and A. De Lucia, “Landfill: An open dataset of code smells with public evaluation,” in *Proceedings of the 12th IEEE/ACM Working Conference on Mining Software Repositories (MSR)*, pp. 482–485, IEEE, 2015.

[61] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, “A large-scale empirical study on the lifecycle of code smell co-occurrences,” *Information and Software Technology*, vol. 99, pp. 1–10, 2018.

[62] F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, “Comparing and experimenting machine learning techniques for code smell detection,” *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[63] L. Amorim, E. Costa, N. Antunes, B. Fonseca, and M. Ribeiro, “Experience report: Evaluating the effectiveness of decision trees for detecting code smells,” in *Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pp. 261–269, IEEE, 2015.

[64] T. Sharma, M. Fragkoulis, and D. Spinellis, “House of cards: Code smells in open-source c# repositories,” in *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 424–429, IEEE, 2017.

[65] A. Ho, A. M. Bui, P. T. Nguyen, and A. Di Salle, “Fusion of deep convolutional and lstm recurrent neural networks for automated detection of code smells,” in *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pp. 229–234, ACM, 2023.

[66] Y. Zhang, C. Ge, H. Liu, and K. Zheng, “Code smell detection based on supervised learning models: A survey,” *Neurocomputing*, vol. 565, p. 127014, 2024.

[67] H. M. Shah, Q. Z. Syed, B. Shankaranarayanan, I. Palit, A. Singh, K. Raval, K. Savaliya, and T. Sharma, “Mining and fusing productivity metrics with code quality information at scale,” in *Proceedings of the 39th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 563–567, IEEE, 2023.

[68] C. Barbosa, A. Uchôa, D. Coutinho, F. Falcão, H. Brito, G. Amaral, V. Soares, A. Garcia, B. Fonseca, M. Ribeiro, and L. Sousa, “Revealing the social aspects of design decay: A retrospective study of pull requests,” in *Proceedings of the 34th Brazilian Symposium on Software Engineering (SBES)*, pp. 364–373, ACM, 2020.

[69] T. Sharma, “Codesplitjava,” 2019. https://github.com/tushartushar/CodeSplitJava.

[70] J. Cohen, “A coefficient of agreement for nominal scales,” *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[71] Y. Yang, X. Hu, Z. Gao, J. Chen, C. Ni, X. Xia, and D. Lo, “Federated learning for software engineering: A case study of code clone detection and defect prediction,” *IEEE Transactions on Software Engineering*, vol. 50, no. 2, pp. 296–321, 2024.

[72] J. Liu, P. Zhou, Y. Hua, D. Chong, Z. Tian, A. Liu, H. Wang, C. You, Z. Guo, L. Zhu, and M. L. Li, “Benchmarking large language models on cmexam - a comprehensive chinese medical exam dataset,” in *Proceedings of the 37th International Conference on Neural Information Processing Systems (NeurIPS)*, vol. 36, pp. 52430–52452, Curran Associates, Inc., 2023.

[73] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, “Graphcodebert: Pre-training code representations with data flow,” in *Proceedings of the 9th International Conference on Learning Representations (ICLR)*, pp. 1–18, OpenReview.net, 2021.

[74] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, “Unixcoder: Unified cross-modal pre-training for code representation,” in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 7212–7225, ACL, 2022.

[75] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, “Deepseek-coder: When the large language model meets programming – the rise of code intelligence,” *arXiv preprint*

arXiv:2401.14196, 2024.

[76] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, and et al., "Starcoder: may the source be with you!," *arXiv preprint arXiv:2305.06161*, 2023.

[77] Y. Liu, L. Gao, M. Yang, Y. Xie, P. Chen, X. Zhang, and W. Chen, "Vuldetectbench: Evaluating the deep capability of vulnerability detection with large language models," *arXiv preprint arXiv:2406.07595*, 2024.

[78] A. Karmakar and R. Robbes, "What do pre-trained code models know about code?," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1332–1336, IEEE, 2021.

[79] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," in *Proceedings of the 35th Conference on Neural Information Processing Systems (NeurIPS)*, pp. 1–16, Curran Associates, Inc., 2021.

[80] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu, and et al., "Automating code review activities by large-scale pre-training," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 1035–1047, ACM, 2022.

[81] W. Tang, M. Tang, M. Ban, Z. Zhao, and M. Feng, "Csgvd: A deep learning approach combining sequence and graph embedding for source code vulnerability detection," *Journal of Systems and Software*, vol. 199, p. 111623, 2023.

[82] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 39–51, ACM, 2022.

[83] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, "Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pp. 654–668, ACM, 2023.

[84] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "Vulrepair: a t5-based automated software vulnerability repair," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 935–947, ACM, 2022.

[85] T.-T. Nguyen, T. T. Vu, H. D. Vo, and S. Nguyen, "An empirical study on capability of large language models in understanding code semantics," *arXiv preprint arXiv:2407.03611*, 2024.

[86] C. Su and C. McMillan, "Distilled GPT for source code summarization," *Automated Software Engineering*, vol. 31, no. 1, p. 22, 2024.

[87] F. Pecorelli, F. Palomba, D. D. Nucci, and A. D. Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," in *Proceedings of the 27th International Conference on Program Comprehension (ICPC)*, pp. 93–104, IEEE, 2019.

[88] J. Yao and M. Shepperd, "Assessing software defection prediction performance: why using the matthews correlation coefficient matters," in *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pp. 120–129, ACM, 2020.

[89] I. Loshchilov, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[90] S. Arshad, S. Abid, and S. Shamail, "Codebert for code clone detection: A replication study," in *Proceedings of the 16th IEEE International Workshop on Software Clones (IWSC)*, pp. 39–45, IEEE, 2022.

[91] S. M. Rabbani, N. A. Gulzar, S. Arshad, S. Abid, and S. Shamail, "A comparative analysis of clone detection techniques on semanticclonebench," in *Proceedings of the 16th IEEE International Workshop on Software Clones (IWSC)*, pp. 16–22, IEEE, 2022.

[92] L. Fan, J. Liu, Z. Liu, D. Lo, X. Xia, and S. Li, "Exploring the capabilities of llms for code change related tasks," *arXiv preprint arXiv:2407.02824*, 2024.

[93] Y. Tay, M. Dehghani, S. Abnar, H. W. Chung, W. Fedus, J. Rao, S. Narang, V. Q. Tran, D. Yogatama, and D. Metzler, "Scaling laws vs model architectures: How does inductive bias influence scaling?," in *Proceedings of the 28th International Conference on Empirical Methods in Natural Language Processing (EMNLP): Findings*, pp. 12342–12364, ACL, 2023.

[94] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *Proceedings of the 36th International Conference on Neural Information Processing Systems (NeurIPS)*, Curran Associates Inc., 2022.

[95] W. Liu, Z.-g. Hu, H.-t. Liu, and L. Yang, "Automated pattern-directed refactoring for complex conditional statements," *Journal of Central South University*, vol. 21, pp. 1935–1945, 2014.

[96] A. Alhefdhi, H. K. Dam, Y. S. Nugroho, H. Hata, T. Ishio, and A. Ghose, "A framework for conditional statement technical debt identification and description," *Automated Software Engineering*, vol. 29, no. 2, p. 60, 2022.

[97] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[98] A.-M. Chaniotaki and T. Sharma, "Architecture smells and pareto principle: a preliminary empirical exploration," in *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR)*, pp. 190–194, IEEE, 2021.

[99] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. de Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly, "Parameter-efficient transfer learning for NLP," in *Proceedings of the 36th International Conference on Machine Learning (ICML)*, vol. 97, pp. 2790–2799, PMLR, 2019.

[100] J. Cao, W. Zhang, and S.-C. Cheung, "Concerned with data contamination? assessing countermeasures in code language model," *arXiv preprint arXiv:2403.16898*, 2024.

[101] G. A. Campbell and P. P. Papapetrou, *SonarQube in action.* Manning Publications Co., 2013.

[102] S. Vidal, H. Vazquez, J. A. Diaz-Pace, C. Marcos, A. Garcia, and W. Oizumi, "Jspirit: a flexible tool for the analysis of code smells," in *Proceedings of the 34th International Conference of the Chilean Computer Science Society (SCCC)*, pp. 1–6, IEEE, 2015.

[103] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aimeur, "Support vector machines for anti-pattern detection," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 278–281, ACM, 2012.

[104] D. Wu, F. Mu, L. Shi, Z. Guo, K. Liu, W. Zhuang, Y. Zhong, and L. Zhang, "ismell: Assembling llms with expert toolsets for code smell detection and refactoring," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1345–1357, ACM, 2024.

[105] J. He, C. Zhou, X. Ma, T. Berg-Kirkpatrick, and G. Neubig, "Towards a unified view of parameter-efficient transfer learning," in *Proceedings of the 10th International Conference on Learning Representations (ICLR)*, pp. 1–15, OpenReview.net, 2022.