An Empirical Study of Parameter-Efficient Fine-Tuning in Code Change Learning and Beyond

Shuo Liu, Jacky Keung, *Senior Member, IEEE*, Zhi Jin, *Fellow, IEEE*, Zhen Yang, *Member, IEEE*, Fang Liu, *Member, IEEE*, and Hao Zhang

***Abstract*—Compared to Full-Model Fine-Tuning (FMFT), Parameter-Efficient Fine-Tuning (PEFT) has demonstrated superior efficacy and efficiency in several code understanding tasks, owing to PEFT's ability to alleviate the catastrophic forgetting issue of Pre-trained Language Models (PLMs) by updating only a small number of parameters. However, existing studies primarily involve static code comprehension, aligning with the pre-training paradigm of recent PLMs and facilitating knowledge transfer, but they do not account for dynamic code changes. Thus, it remains unclear whether PEFT outperforms FMFT in task-specific adaptation for code-change-related tasks. To address this question, we examine four prevalent PEFT methods (i.e., AT, LoRA, PT, and PreT) and compare their performance with FMFT across seven popular PLMs. In experiments, two widely studied code-change-related tasks, i.e., Just-In-Time Defect Prediction (JIT-DP) and Commit Message Generation (CMG) are involved, demonstrating that the four PEFT methods can surpass FMFT on JIT-DP but only exhibit comparable performances at best on CMG in common scenarios. While in cross-lingual and low-resource scenarios, they exhibit relative superiority. Afterward, a series of probing tasks from both static and dynamic perspectives are conducted in this paper, offering detailed explanations for the efficacy of PEFT and FMFT. Inspired by the distinctive advantages of PEFT and FMFT in their layer-wise probing results, we propose Pasta$_K$, a self-adaPtive efficient layer-specific tuning framework for PLMs in code change learning, which combines FMFT and PEFT during the domain adaptation according to the guidance of probing results. Experiments in the CMG task demonstrate that Pasta$_K$ surpasses diverse PEFT methods in effectiveness. Even, Pasta$_K$ outperforms FMFT by 1.48%, 3.21%, and 1.87% at most in terms of BLEU, Meteor, and Rouge-L, while saving 26.26% and 20.65% in terms of training time and computational memory compared with FMFT.**

***Index Terms*—Adapter tuning, low-rank adaptation, prompt tuning, prefix tuning, commit message generation, just-in-time defect prediction, CodeBERT, GraphCodeBERT, PLBART, UniXcoder, CodeT5, CodeT5+, Qwen2.5-Coder.**

I. Introduction

PRE-TRAINED Language Models (PLMs), e.g., CodeBERT [1] and CodeT5 [2], have been extensively utilized in various software engineering tasks, including code generation [3], [4] and code comprehension [5], [6], having achieved notable advancements [7]. These models follow a prevalent paradigm where they are initially pre-trained on large-scale corpora using Masked Language Modeling (MLM) or Next Token Prediction (NTP) [8], and subsequently fine-tuned for specific downstream tasks. Nonetheless, Full-Model Fine-Tuning (FMFT) for PLMs or even LLMs prohibitively relies on enormous computational resources, rendering it unsuitable for most users. Therefore, a series of Parameter-Efficient Fine-Tuning (PEFT) methods [9], [10], [11], [12], updating only 0.01%-5% of the whole parameters while keeping the others frozen, have been proposed in recent years and gained much attention [13], [14], [15].

Previous studies [16], [17] revealed the superiority of PEFT against FMFT in code search, code clone, and code summarization tasks. Because PEFT greatly alleviates the catastrophic forgetting problem, it can efficiently harness the pre-trained knowledge for the downstream tasks [18]. However, it is noted that most of the investigation objects of the above literature are static code comprehension tasks, where code snippets are learned in only a single status. For example, code summarization, shown in Fig. 1(a), is a typical task of this kind, whose input is a code snippet of a single status while producing a code comment as an output. Apparently, this is consistent with the paradigm that PLMs learned during their pre-training phase. As such, we conjecture that adapting PLMs to static code comprehension tasks inherently needs fewer parameter tuning. In contrast, dynamic code comprehension tasks (e.g., Fig. 1(b) and (c)), such as Commit Message Generation (CMG) and Just-In-Time Defect Prediction (JIT-DP), differ significantly from the

Received 10 February 2025; revised 31 October 2025; accepted 12 November 2025. Date of publication 27 November 2025; date of current version 9 January 2026. This work was supported in part by the General Research Fund of the Research Grants Council of Hong Kong and the research funds of the City University of Hong Kong under Grant 6000796, Grant 9229109, Grant 9229098, Grant 9220103, and Grant 9229029, in part by the National Natural Science Foundation of China Grant 62302021, Grant 62192731, Grant U24B20149, and Grant 62502283, in part by the Natural Science Foundation of Shandong Province under Grant ZR2024QF093, and in part by the Young Talent of Lifting Engineering for Science and Technology in Shandong, China. Recommended for acceptance by C. McMillan. *(Corresponding author: Zhen Yang.)*

Shuo Liu and Jacky Keung are with the Department of Computer Science, City University of Hong Kong, Hong Kong 999077, China (e-mail: sliu273-c@my.cityu.edu.hk; Jacky.Keung@cityu.edu.hk).

Zhi Jin is with the Key Laboratory of High Confidence Software Technologies (PKU), MOE, Peking University, Beijing 100871, China, and also with the School of Computer Science, Wuhan University, Wuhan 430072, China (e-mail: zhijin@pku.edu.cn).

Zhen Yang and Hao Zhang are with the School of Computer Science and Technology, Shandong University, Qingdao 266237, China (e-mail: zhenyang@sdu.edu.cn; haozhang@mail.sdu.edu.cn).

Fang Liu is with the School of Computer Science and Engineering, Beihang University, Beijing 100191, China (e-mail: fangliu@buaa.edu.cn).

Digital Object Identifier 10.1109/TSE.2025.3637335

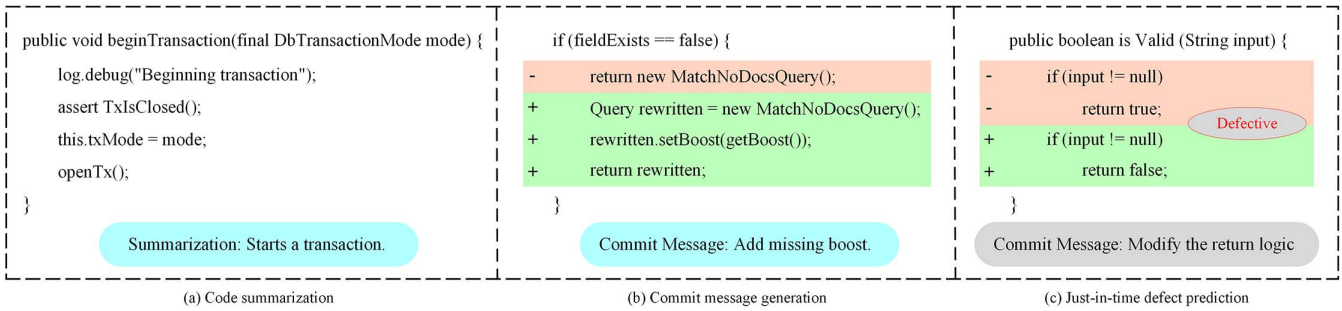


Fig. 1. Representative code intelligence tasks related to static and dynamic code snippets.

above static ones. They contain code changes (i.e., code *diff*s) within the model inputs, where both old and new versions of code snippets co-occur, inadvertently enlarging the transferring gap between the pre-training and adaptation phases [19], [20]. The above code change-related tasks are ubiquitous in developers' daily work and are important for project maintenance, as developers need to debug with code changing and review code *diff*s submitted by other developers for collaboration. As such, it is necessary to explore the effect of PEFT on code change learning, thereby seeking the potential for further improvement, and probing its application discrepancies against static code learning.

In this paper, we choose four mainstream PEFT methods for experiments, namely Adapter Tuning (AT) [9], [21], [22], Low-Rank Adaptation (LoRA) [10], Prompt Tuning (PT) [12], and Prefix Tuning (PreT) [11]. AT introduces adapter modules, each of which is a bottleneck structure, and inserts them between layers of PLMs. In the fine-tuning stage, the parameters of the original PLMs are fixed, and only adapter modules are adjusted. LoRA injects trainable rank decomposition matrices into PLMs to substitute the original pre-trained weights, which can amplify some hidden features encoded during pre-training [10]. PT and PreT leverage the power of prompting in eliciting PLMs' pre-trained knowledge [23]. The former adds trainable vectors as prompts to the original input, but the latter introduces extra trainable parameters to the hidden states. To evaluate their performance on code change learning, we conduct experiments for JIT-DP [24] and CMG [25]. The former is a classification task aiming to predict whether a code change is defect-prone or not, and the latter is a generation task targeting to automatically generate a commit message given changed code snippets.

Specifically, (i) we first compare the performance of PEFT methods to FMFT and State-Of-The-Art (SOTA) approaches with full-data settings, namely the common scenario. Experimental results show that, in the JIT-DP task, all four PEFT methods can surpass FMFT and even SOTA approaches when incorporating extra expert features, with AT and LoRA obtaining improvements of 5.08% and 9.75% in F1 over SOTA baselines. In the CMG task, AT and LoRA achieve comparable performance to FMFT and SOTA baselines, while PT and PreT perform worse. Given their lower training time and memory overhead, AT and LoRA remain practical and acceptable. (ii) Besides, to evaluate how well PEFT methods transfer knowledge and generalize across Programming Languages (PLs), we explore their capabilities in the cross-lingual scenario, where PLMs are fine-tuned in one PL but are tested in another. Evaluation results show that PEFT methods perform either better or comparable to FMFT. (iii) To understand how well PEFT performs in low-resource scenarios, we scaled down the fine-tuning datasets while keeping the testing samples the same for experiments. Evaluation results reveal that almost all PEFT methods fall short of FMFT on the CMG task, whereas they exhibit overall superiority or are on par with FMFT on the JIT-DP task in the w/ EF setting. (iv) Furthermore, we utilize three probing tasks, namely invalid TYPe detection (TYP), Code Change Match (CCM), and Line Type Prediction (LTP), to investigate the layer-wise encoded code semantics, expecting to make an explanation for the performance of PEFT and FMFT. TYP relates to static code semantics, while CCM and LTP are associated with dynamic code semantics from global and local perspectives, respectively. The average probing results are consistent with previous findings and explain the efficacy of PEFT techniques and FMFT. However, PEFT and FMFT demonstrate distinctive code-change learning abilities among different layers of PLMs.

Inspired by the distinctive advantages of PEFT and FMFT in their layer-wise probing results, we propose **Pasta$_K$**, which stands for self-adaPtive efficient layer-specific tuning for pre-trained models in code change learning, to combine FMFT and PEFT during the domain adaptation. Since the combination and assignment of PEFT and FMFT layers are based on the results of probing tasks, this reflects the self-adaptability of Pasta$_K$ to experimental data. Here, K is a hyperparameter representing the number of FMFT layers, i.e., the resource constraint. In experiments, K is set to 0, 3, 6, 9, and 12 for evaluation. According to the layer-wise probing results, Pasta$_K$ allocates PEFT and FMFT across different layers of PLMs for tuning, thereby integrating their respective strengths. We conduct experiments using all four PEFT methods on the CMG task to verify their effectiveness. Experimental results show that Pasta$_K$ significantly outperforms diverse PEFT methods. Even compared with FMFT, Pasta$_K$ can surpass it by 1.48%, 3.21%, and 1.87% at most in terms of BLEU, Meteor, and Rouge-L, respectively. Meanwhile, Pasta$_K$ remains efficient, saving

26.26% of training time and 20.65% of computational memory compared with FMFT, indicating the practical value of Pasta$_K$.

The major contributions of this paper are as follows:

1. To the best of our knowledge, this paper is the first to explore the performance of PEFT on code-change-related tasks.
2. We conduct extensive experiments, involving four PEFT methods, seven PLMs, and two specific code-change-related tasks. We also explore their performance in the cross-lingual and low-resource scenarios.
3. We propose two code-change-related probing tasks to investigate the encoded dynamic code semantics and construct corresponding datasets.
4. Inspired by our probing results, we propose an approach, Pasta$_K$, that combines FMFT and PEFT during domain adaptation and conduct extensive experiments to demonstrate its efficacy and efficiency, thereby showing its practical value.
5. We open-source the replication package of this work for future research and application.[1]

This work is an extension of our SANER 2024 paper [26]. Compared to the previous version, we newly added two PEFT methods for experiments, namely PT and PreT, in Section II. Accordingly, we conducted experiments for PT and PreT on all four original research questions, and the experimental results and analyses were included in Section IV-A - Section IV-D. Besides, we involved an additional research question (RQ5) about the new approach named Pasta$_K$ in Section IV-E. The experiments illustrate that Pasta$_K$ can significantly improve the performance of PEFT methods. Even when compared with FMFT, Pasta$_K$ can perform better while consuming less training time and memory size.

II. Background

A. Full-Model Fine-Tuning

Pre-training and fine-tuning have become the dominant paradigm for applying PLMs to specific downstream tasks, and have shown promising results on many code-related tasks [27], [28], [29], [30]. In the pre-training stage, PLMs are trained from scratch to learn general-purpose representations, and in the fine-tuning stage, PLMs utilize the pre-trained parameters as initialization to adapt them to downstream tasks. Formally, given a downstream task-specific dataset X and corresponding labels Y, FMFT initializes PLMs to pre-trained weights Θ_0 and updates them to Θ in a supervised manner, which can be formulated as:

$$\Theta = \Theta_0 + \Delta\Theta = \arg\max_{\Theta} Pr(Y|X;\Theta) \tag{1}$$

where Pr represents the likelihood of Y and $\Delta\Theta$ represents the adjustment of parameters. Considering that PLMs are Transformer-based architecture [31], FMFT is illustrated in Fig. 2(a). The parameters of the core block of the Transformer, including multi-head attention, feed-forward layer, and layer normalization, as well as the projection matrices W_Q, W_K, and W_V, all are part of Θ and are trainable in FMFT for PLMs.

[1]https://github.com/ishuoliu/PeftCC

B. Adapter Tuning (AT)

AT is a representative method of PEFT, which adds extra compact modules (adapters) to every transformer layer [9], [21], [22]. Fig. 2(b) shows the standard architecture of AT [9], where adapter modules are added twice after the multi-head attention and the feed-forward layer. Supposing that adapter modules are fed d-dimensional features, they first project features into dimension m, $m \ll d$, then apply a nonlinearity, and finally project back to dimension d. The two projection layers construct a bottleneck structure. Specifically, adapter modules are initialized randomly, and the target of AT is to train the added modules and corresponding following layer normalizations. Its tuning objective can be defined as:

$$\Phi_a = \arg\max_{\Phi_a} Pr(Y|X;\Theta_0,\Phi_a) \tag{2}$$

where Φ_a is the adapter parameter, and Pr represents the likelihood. X and Y are a task-specific dataset and corresponding outputs. PLMs' pre-trained weights Θ_0 are frozen here. During AT, around 3% - 5% of the whole parameters are trainable, which is considerably fewer than FMFT.

C. Low-Rank Adaptation (LoRA)

Different from AT, which adds extra trainable parameters, LoRA utilizes the idea of reparameterization to learn the parameters' updation by low-rank matrices [32]. As illustrated in Fig. 2(c), when applying LoRA to the transformer architecture, it is limited to only adapting the attention modules, and specifically, adapting weight matrices W_Q and W_V [10]. For a pre-trained weight matrix $W \in \mathbb{R}^{d\times d}$, LoRA decomposes its updation into two low-rank matrices that satisfy $\Delta W = BA$, where $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times d}$, $r \ll d$. Supposing $\Delta\Theta$ is a task-specific parameter modification, it can be further encoded by the LoRA parameter Φ_l as $\Delta\Theta(\Phi_l)$. The tuning objective is then transferred to optimize Φ_l as:

$$\Phi_l = \arg\max_{\Phi_l} Pr(Y|X;\Theta_0 + \Delta\Theta(\Phi_l)) \tag{3}$$

where X and Y are the dataset and corresponding outputs. Pr represents the likelihood of Y. Θ_0 is also fixed here. Due to LoRA being only inserted into weight matrices W_Q and W_V, as well as the dimension $r \ll d$, the trainable parameters of LoRA can be reduced to 1%.

D. Prompt Tuning (PT)

Rather than inserting tiny modules inside PLMs as the above two PEFT methods, PT prepends a series of prompt tokens P to the input X, formulating a new input $[P;X]$ for model training [12]. The strength of PT lies in its ability to steer PLMs toward effectively leveraging the rich knowledge acquired during the pre-training phase simply by optimizing prompts, while also demonstrating robustness even under data-scarce scenarios. It has been investigated in lots of static code comprehension tasks

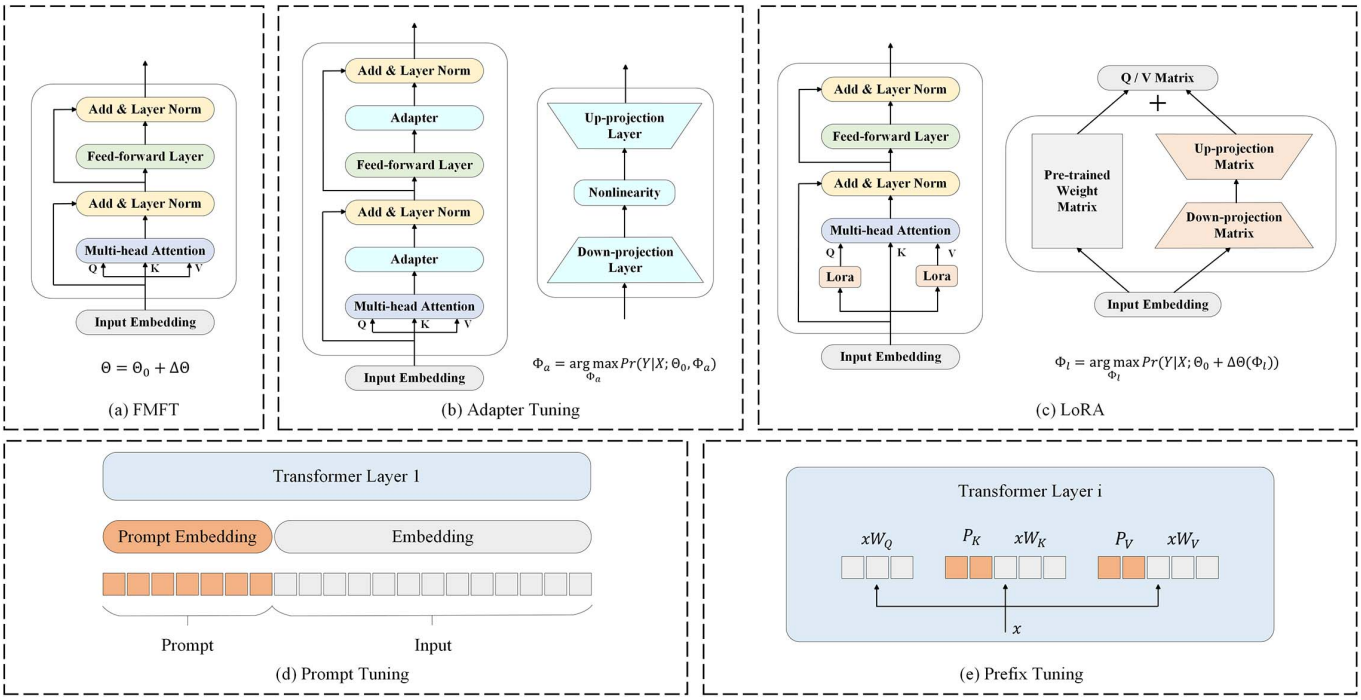


Fig. 2. Illustrations on FMFT, AT, LoRA, PT, and PreT.

[33], [34], and we extend its exploration to the realm of dynamic code comprehension tasks. As shown in Fig. 2(d), prompt tokens are inserted into the first Transformer layer only. When optimizing prompt tokens P, PLMs' pre-trained weights Θ_0 are fixed, and the trainable parameters are only about 0.03%. The tuning objective can be formulated to maximize the likelihood of Y, which represents the expected outputs of X:

$$P = \arg\max_{P} Pr(Y|[P;X];\Theta_0) \tag{4}$$

Considering that PT is an effective approach to elicit PLMs' pre-trained knowledge [23] and has been widely studied in recent literature [15], [20], [33], [35], we include it for exploration on code change learning.

E. Prefix Tuning (PreT)

Similar to PT, PreT [11] is also a prevalent prompt-based method that has been widely studied [13], [15], [20], [36]. Unlike PT, PreT inserts trainable prefix vectors to each Transformer layer, thereby enhancing the expressiveness of PLMs when activating their previously acquired knowledge. The trainable vectors are introduced to the original key vector and value vector, which are used for the calculation of multi-head attention. We include PreT in our exploration since it is another comparable counterpart to AT and LoRA, both of which inject trainable modules into PLMs. Fig. 2(e) shows an illustration of PreT. In each transformer layer, there are two sets of prefix vectors P_K and P_V that are concatenated with xW_K and xW_V, respectively. Here x is the input vector. W_K and W_V are the original weight matrices. xW_K and xW_V are the original key vector and value vector. Only the prefix vectors are trainable, which are about 0.5%, and PLMs' pre-trained weights Θ_0 are fixed. The tuning objective can be defined to maximize the likelihood of the input X's corresponding output Y:

$$[P_K;P_V] = \arg\max_{P_K,P_V} Pr(Y|X;\Theta_0,P_K,P_V) \tag{5}$$

F. Training-Free Methods

The above-introduced methods need parameter updating of PLMs for domain adaptation. Here, we present two training-free methods that PLMs also use for baseline comparison. The first one is Zero-shot Learning (ZL). It bypasses the fine-tuning process and directly guides PLMs/Large Language Models (LLMs) to perform prediction or generation on downstream tasks without learning from any training samples. ZL is more pervasive for LLMs owing to their powerful emerging ability [37], [38]. The second one is In-Context Learning (ICL), which also does not require the fine-tuning process; instead, it guides PLMs/LLMs to perform specific downstream tasks by providing examples within the input. Recently, more and more studies have proved its effectiveness, especially when the given examples resemble the target problem [27], [39].

III. Experimental Setup

A. Research Questions

This paper investigates the following research questions about four widely used PEFT methods, exploring their efficacy and efficiency.

RQ1: How do the four PEFT methods perform compared with FMFT and SOTA approaches? In this RQ, we compare the performance of FMFT, AT, LoRA, PT, and

PreT on two widely studied code-change-related tasks, namely Just-In-Time Defect Prediction (JIT-DP) and Commit Message Generation (CMG). The former is a classification task, and the latter is a generation task. For JIT-DP, we apply the FMFT and four PEFT methods to six popular PLMs, namely CodeBERT [1], GraphCodeBERT [6], PLBART [40], UniXcoder [41], CodeT5 [2], and CodeT5+ [42], to conduct comprehensive comparisons and explore the generalization abilities of PEFT on diverse PLMs. As a common workaround, classification tasks normally rely on encoders appended with a linear layer for category prediction [19], [43], [44]. Thus, we select only encoder-only and encoder-decoder PLMs for experiments, where the latter's encoder parts are decomposed for usage. Besides, various task-specific SOTA approaches are also involved for comparison [19], [45], [46], [47]. For CMG, we apply FMFT and PEFT methods on CodeT5, CodeT5+, and Qwen2.5-Coder [48] for their strong capabilities in generation tasks. Here, we only select encoder-decoder and decoder-only PLMs for experiments, because encoder-only PLMs do not have generation ability due to their pre-training paradigm. Additionally, we incorporate ZL and ICL as baseline methods in the two tasks to facilitate a more comprehensive analysis.

RQ2: How capable are the four PEFT methods in the cross-lingual scenario compared with FMFT? In the cross-lingual scenario, practitioners normally fine-tune PLMs in one PL of rich samples and apply them in another scarce one for inference. This is a common but resultful approach to relieve the problem of data scarcity [49]. Regardless of static or dynamic code comprehension tasks, cross-lingual scenarios are pervasive with widespread studies [20], [33], [50], [51]. Because there are massive domain-specific PLs that contain very few available training samples in practice. At the time the studies [30], [52] were released, 8,945 PLs have been invented, while Meyerovich et al. [53] found that the top 20 PLs account for 95% of the project repositories, showing that most of the PLs are niche. Therefore, exploring the performance of PEFT methods in the cross-lingual scenario is valuable for practical guidance. To do this, we carry out experiments with FMFT, AT, LoRA, PT, and PreT on CodeT5 and select the CMG task for evaluation, as it involves code change samples from five PLs.

RQ3: How capable are the four PEFT methods in the low-resource scenario compared with FMFT? The performance of FMFT prohibitively relies on the scale of downstream data [12], [35], [54]. However, large-scale datasets are usually rare. In addition, fine-tuning in the low-resource scenario means that PLMs can learn task-specific knowledge quickly, which meets practitioners' usual expectations. Considering the difference in dataset sizes between the two tasks, we randomly sample 100, 500, 1,000, 2,500, and 5,000 training data for JIT-DP, and 1,000, 5,000, and 10,000 per programming language for CMG. To mitigate the impact of data sampling variability, we conduct three independent experiments using different random seeds and report the average results of the three runs. We still evaluate the models on the original validation sets and testing sets.

RQ4: What kind of knowledge is encoded by the four PEFT methods? To explore what benefits PEFT methods can bring about, we utilize probing tasks, which have been extensively used in the NLP field [55], [56], to understand the encoded semantic information. We employ three probing tasks, invalid TYPe detection (TYP) for static semantic-level information [5], as well as Code Change Match (CCM) and Line Type Prediction (LTP), which are proposed by us to probe the dynamic semantic-level information from the global and local perspectives, respectively. The three probing tasks are introduced in detail in section III-B. We mainly investigate what kind of knowledge is encoded by the four PEFT methods, as well as making comparisons with FMFT and ZL, thereby providing explanations for their performance.

RQ5: How to combine the superiority of PEFT and FMFT for better domain adaptation? Based on the experimental results of the aforementioned research questions, we investigate more effective alternatives to the PEFT methods. With much fewer trainable parameters, it is difficult for PEFT to surpass FMFT too much, even worse in some complicated tasks, such as the CMG task. It is observed by no matter previous studies [10], [36] or our experimental evaluation in Section IV-A. Therefore, inspired by our layer-wise probing results, we explore whether there is a brand-new domain adaptation method that can integrate the advantages of PEFT and FMFT together, improving the performance while keeping efficiency.

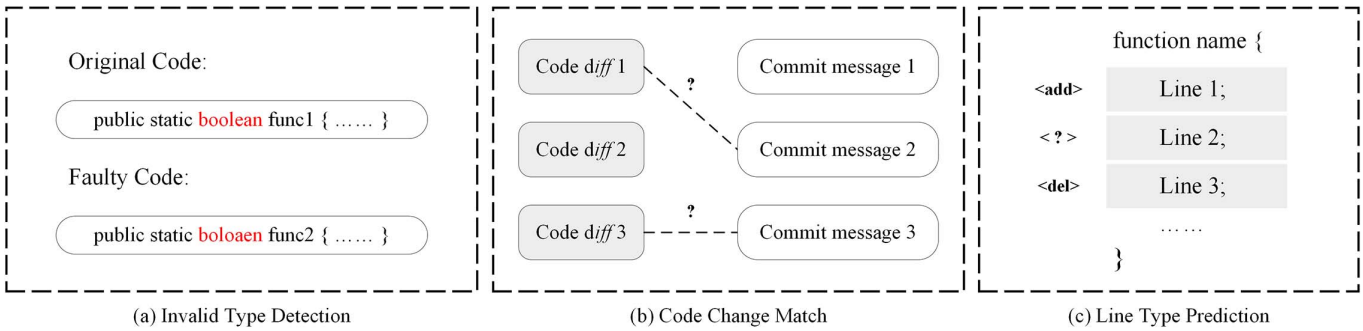
B. Tasks and Datasets

1) Code-Change-Related Tasks: To evaluate the performance of PEFT methods in code change learning, we consider two emerging code-change-related tasks, just-in-time defect prediction and commit message generation, which have been popularly researched in recent years [24], [25], [39], [57].

Just-In-Time Defect Prediction: The change of code may damage software quality, so it is crucial to discover defects as early as possible [24]. JIT-DP aims to identify defective code changes when they are just committed, and return judgements. It can be seen as a binary classification task since the outputs are two classes, namely defective or not.

The experimental dataset we used in JIT-DP is JIT-Defects4J [47], which is extended from LLTC4J [58] with extra buggy commits. In addition, for each code change, JIT-Defects4J extracts the 14 change-level Expert Features (EF) proposed by Kamei et al. [59], which can measure the code change from five dimensions, i.e. diffusion, size, purpose, history, and experience. It is widely adopted in previous studies [19], [43], [47], [60]. Following their setup, we evaluate different methods in two settings, one for directly using the learned code change representations to predict, and another for incorporating the expert features.

Commit Message Generation: Commit message summarizes the intent and content of a code change, which is helpful for developers to understand programs quickly in software maintenance [25], [57]. However, writing high-quality commit messages is time-consuming, and thus many are left empty [61]. Therefore, taking changed codes as inputs, the CMG task aims to generate commit messages automatically and has become a hot topic in the software engineering domain.



(a) Invalid Type Detection (b) Code Change Match (c) Line Type Prediction

Fig. 3. Illustrations of probing tasks.

TABLE I
DATASET STATISTICS

Tasks	Datasets	Training	Validation	Test
JIT-DP	JIT-Defects4J	16,374	5,465	5,480
CMG	Java	160,018	19,825	20,159
	C#	149,907	18,688	18,702
	C++	160,948	20,000	20,141
	Python	206,777	25,912	25,837
	JavaScript	197,529	24,899	24,773
Probing Tasks	TYP	600	200	200
	CCM	600	200	200
	LTP	600	200	200

In the CMG task, we choose the Multi-language Commit Message Dataset (MCMD) for experiments [62]. MCMD contains five PLs, including Java, C#, C++, Python, and Javascript. For each PL, it collects the top 100 starred projects from GitHub and randomly retains 450,000 commits. It has been a widely used large-scale dataset for the CMG task and has been adopted by numerous prior studies [19], [57], [63], [64]. Following the common operation, noisy data containing multiple files or files that cannot be parsed are filtered out. The statistics of MCMD are shown in Table I.

2) Probing Tasks: Probing tasks are auxiliary diagnostic tasks that are used to understand what specific property is encoded in the language model weights. It has been widely utilized in NLP research [65]. In this paper, we employ three probing tasks that are related to code properties from different aspects. For their datasets, we split them into the training set, validation set, and testing set by the ratio of 6/2/2 uniformly.

Invalid TYPe detection (TYP), proposed by Karmakar et al. [5], aims to measure the static code semantics understood by PLMs. As Fig. 3(a) shows, it falsifies the data types of some code snippets deliberately, and expects PLMs to distinguish the invalid samples from the others. We introduce the probing task here to explore whether PLMs still encode static code semantics when they are fine-tuned in code-change-related tasks. We also adopt the dataset containing 1,000 samples from Karmakar et al. [5]. It is gathered from 50K-C [66], a dataset of compilable Java projects, and balances the valid and invalid classes.

Code Change Match (CCM), probing the dynamic semantic-level information, is proposed by us. Different from TYP, designed for code semantics in one code snippet, CCM takes a pair of changed codes and a commit message as inputs, inferring whether the commit message corresponds to the given code change. It is shown in Fig. 3(b). If the judgment is correct, it means PLMs can identify the code change semantics from a global perspective. We construct a dataset for CCM from FIRA [67], which collects commits from the top 1,000 popular Java projects in GitHub. Similar to the dataset of TYP, we keep 500 correct matches and randomly produce 500 false matches.

Line Type Prediction (LTP), proposed by us as well, is also to determine whether PLMs can encode dynamic semantic-level information. As code changes are composed of added, kept, and deleted lines, LTP aims to predict the line type when given the surrounding context, as illustrated in Fig. 3(c). Considering LTP is designed for predicting specific line types, it measures PLMs' ability to comprehend the semantics of code changes from a local perspective. We also build the dataset for LTP from FIRA [67]. For each commit, to measure the perception of code changes, we only retain the added and deleted lines, then randomly mask a line type for prediction. The dataset also incorporates 1,000 samples, and Table I demonstrates its statistics.

Although probing tasks could be designed with finer granularity, for example, by evaluating whether PLMs can distinguish between adding a variable declaration, modifying a return value, or deleting an unnecessary function call, however, such complicated classification tasks might intensify interpretability problems [68], deviating from the original intent of probing tasks, i.e., focusing on simple PL properties of code snippets [5]. An alternative solution is to deconstruct the above multi-category classification into several different probing tasks for more fine-grained exploration. As such, we leave them for future work and begin with the simple probing tasks proposed in this section for analysis.

C. Pre-Trained Language Models and Baselines

There are a total of seven PLMs that we use to evaluate the performance of FMFT and the four PEFT methods. In the JIT-DP task, we conduct experiments on six PLMs, where only encoders are used for the purpose of code change representation. The selected PLMs have all been widely used and extensively studied in code-related tasks [17], [36], which facilitates the generalization and comparability of our experimental findings. We include CodeT5+ here because it is a more recent

model whose encoder can be used independently, enabling fair and consistent experimentation with other PLMs in the JIT-DP task. For the CMG task, we choose CodeT5, CodeT5+, and Qwen2.5-Coder as the backbone model. The selection of CodeT5 and CodeT5+ is motivated by the need to align with the existing SOTA approach [19], while Qwen2.5-Coder is included due to its more recent release, strong performance in generation tasks, and a parameter scale comparable to that of the other PLMs in our study. The specific introduction for each model is listed below.

- **CodeBERT** [1]: It is an encoder-only PLM with 125M parameters, pre-trained on CodeSearchNet [69] which contains 2M bimodal data.
- **GraphCodeBERT** [6]: It has the same architecture and parameter size as CodeBERT, while pre-trained with edge prediction and node alignment to learn from data flow.
- **PLBART** [40]: It is pre-trained on Java and Python datasets, as well as natural language descriptions. It is an encoder-decoder architecture that has 140M parameters.
- **UniXcoder** [41]: It is a unified encoder-decoder PLM with 125M parameters, and can be flexibly modified to encoder-only or decoder-only architecture.
- **CodeT5** [2]: It contains 220M parameters, and is a representative PLM of encoder-decoder architecture for its good performance in generation tasks.
- **CodeT5+** [42]: It is based on CodeT5 and enhanced with a mixture of pretraining objectives to flexibly adapt to various downstream tasks. We use the 220M-sized variant to maintain consistency with CodeT5.
- **Qwen2.5-Coder** [48]: It is a new series of decoder-only language models demonstrating top-tier performance in coding tasks. We employ the 500M-parameter model in this paper.

We also compare the performance of FMFT and PEFT methods with the following baselines.

- **JITLine** [45]: It is a baseline of JIT-DP, which represents code changes by bag-of-tokens features and constructs classifiers like SVM to predict defective commits.
- **JITFine** [47]: It uses CodeBERT as an encoder, and incorporates the encoded code change representations with extra expert features, achieving a substantial improvement in the JIT-DP task.
- **CC2Vec** [46]: It is a baseline of JIT-DP, which utilizes a hierarchical attention network and emphasizes modeling the correlation between removed codes and added codes.
- **CodeReviewer** [44]: It is proposed for code review activities, but can be extended to the CMG task. It proposes four pre-training tasks to improve CodeT5's [2] capacity for understanding code changes.
- **CCT5** [19]: It is the state-of-the-art baseline for both JIT-DP and CMG. It also pre-trains CodeT5 [2] with code-change-related corpus and objectives.

D. Evaluation Metrics

1) Just-In-Time Defect Prediction: Owing to the imbalance learning nature of the JIT-DP task, prior works [19], [47] normally use the F1-score and the Area Under the receiver operating characteristic Curve (AUC) as the evaluation metrics. Hence, we follow their settings for evaluation. F1-score combines the precision and recall scores of a model, and AUC reflects the distinguishing ability of models between positive and negative classes. In the JIT-DP task, we treat defect-prone code changes as positive samples.

2) Commit Message Generation: For the CMG task, we evaluate the generated commit messages using three metrics: BLEU [70], Meteor [71], and Rouge-L [72]. BLEU calculates the n-gram precision between the generated text and ground truth text. Meteor takes into account the precision, recall, and fluency of the generated text. Rouge-L focuses on the longest common subsequence so that it evaluates more about the word order. Specifically, for the BLEU metric, we choose a smoothed BLEU-4 score for evaluations in this paper.

3) Probing Tasks: Due to probing tasks being classification tasks and their class distributions being balanced, we use accuracy as the evaluation metric, which measures the correctness of predictions.

E. Implementation Details

The experiments were conducted on an Ubuntu GPU server with four RTX 3090 24GB GPUs. Our code is implemented by the deep learning framework PyTorch[2]. The PLMs we used are from Huggingface[3], and we keep their default hyperparameter settings. For the AT, we implement it with the OpenDelta Library[4], and we set the intermediate dimension m to 128 because it strikes an effective balance between performance and parameter efficiency. LoRA, PT, and PreT are implemented by the PEFT Library[5]. Based on prior findings on optimal LoRA rank [10], we set the LoRA rank $r = 8$. After sweeping over prompt lengths 10, 20, 30, 40, 50, we find the optimal prompt length for the latter two methods to be 50. An elaborate analysis of these PEFT-related parameters will be discussed in section IV-A4. In the JIT-DP task, we adjust the learning rate from 5e-5 to 5e-2 for all PLMs. In the CMG task, we set the learning rate from 5e-5 to 1e-2 for different methods after a grid search. Besides, we set the maximum training epoch to 10, and adopt early stopping with a patience of 5.

IV. Experimental Results

A. RQ1: Quantitative Evaluation

1) Just-In-Time Defect Prediction: We first evaluate the performance of the four PEFT methods in the JIT-DP task. We employ six prevalent PLMs to exhibit comprehensive comparisons. Following previous studies [19], [47], we also conduct experiments on two settings. In the first setting (w/o EF), encoded code change representations are extracted and then fed into a linear classifier, predicting defectiveness or not. The second setting (w/ EF) combines the encoded representations with

[2]https://pytorch.org/
[3]https://huggingface.co/models
[4]https://opendelta.readthedocs.io/en/latest/
[5]https://huggingface.co/docs/peft/index

extra expert features [59], and the concatenated feature vectors are used to predict. Table II shows the detailed evaluation results. Digits in bold denote the best value among all results. Table cells in a light grey background denote the best result among different backbone models under the same approach. Because JITLine and CC2Vec do not utilize expert features in their original papers, their performances in the second setting are omitted.

When comparing the performance of FMFT with AT and LoRA, it can be noticed that no matter in which setting, their average performances are always better than FMFT. To be specific, in the first setting (w/o EF), AT obtains an improvement of 3.17% and 1.53% in terms of F1 and AUC, while LoRA increases by 3.17% and 1.29%. When incorporating extra expert features, the improvements become significant. AT surpasses fine-tuning by 16.43% and 2.06% for F1 and AUC, respectively. LoRA increases 21.60% and 2.75% accordingly. However, as for the PT and PreT, they underperform FMFT in the first setting. As Table II indicates, when excluding expert features, PT decreases 11.11% and 7.63% in terms of F1 and AUC, while PreT declines 6.88% and 5.99% accordingly compared with FMFT. In the second setting, which incorporates extra expert features, it can be observed that CodeBERT, GraphCodeBERT, and UniXcoder perform better than FMFT when using PT and PreT, while the other three encoder-decoder models perform worse. Since only the encoder is utilized in these encoder-decoder models, they have fewer trainable parameters and limited representational capacity, which results in inferior performance when applying PT and PreT. Meanwhile, this highlights that the parameter scale of PT and PreT is insufficient to fully handle the complexities of dynamic code comprehension tasks. However, their performance can improve when augmented with additional expert features to aid PLMs' understanding of the tasks.

The above results indicate that AT and LoRA show their effectiveness in the JIT-DP task, but PT and PreT may underperform FMFT as their representation abilities degrade when only updating quite a few parameters. Besides, when introducing extra expert features, the results also show that the straightforward features are prone to being utilized by PEFT methods, leading to noteworthy improvements. A potential explanation is that PEFT methods with very few parameter tuning are more capable of learning straightforward features like expert features, thereby obtaining substantial improvement. In addition, it can be observed that AT and LoRA consistently show their improvements in diverse PLMs, which manifests their generalization abilities.

When compared with other baselines, AT and LoRA achieve state-of-the-art results that reach 0.496 and 0.518 in terms of F1 in the setting with extra expert features. They surpass CCT5 [19], the current SOTA method, by 5.08% and 9.75%, respectively. Although the average performances of PT and PreT are lower than CCT5, they still can achieve surprising results when employing UniXcoder, reaching 0.517 and 0.566 for the F1 metric, and also performing better than CCT5. Considering that CCT5 has been pre-trained on code-change-related corpus with various objectives, it indicates the prominent advantage of the four PEFT methods because they only need to fine-tune a very small portion of the parameters. For ZL and ICL, they show substantially worse performance compared to PEFT methods and FMFT, indicating that fine-tuning remains essential for the effective application of PLMs.

TABLE II
EVALUATION RESULTS OF JIT-DP ON THE JIT-DEFECTS4J DATASET

Models		w/o EF		w/ EF	
		F1	AUC	F1	AUC
End-to-end	JITLine	0.261	0.802	-	-
	JITFine	0.375	0.856	0.431	0.881
	CC2Vec	0.248	0.791	-	-
Pre-trained	CCT5	**0.451**	**0.871**	0.472	0.882
FMFT	CodeBERT	0.382	0.849	0.419	0.867
	GraphCodeBERT	0.390	0.869	0.361	0.861
	PLBART	0.329	0.834	0.412	0.869
	UniXcoder	0.401	0.847	0.480	0.889
	CodeT5	0.398	0.859	0.433	0.878
	CodeT5+	0.366	0.851	0.452	0.878
	Avg.	0.378	0.852	0.426	0.874
LoRA	CodeBERT	0.380	0.860	0.512	0.899
	GraphCodeBERT	0.379	0.869	0.523	0.898
	PLBART	0.393	0.862	0.512	0.890
	UniXcoder	0.388	0.852	0.527	**0.907**
	CodeT5	0.376	0.867	0.519	0.895
	CodeT5+	0.425	0.867	0.515	0.898
	Avg.	0.390	0.863	0.518	0.898
AT	CodeBERT	0.408	0.864	0.502	0.894
	GraphCodeBERT	0.377	0.866	0.525	0.897
	PLBART	0.377	0.858	0.494	0.890
	UniXcoder	0.402	**0.871**	0.515	0.901
	CodeT5	0.404	0.866	0.522	0.892
	CodeT5+	0.371	0.862	0.418	0.875
	Avg.	0.390	0.865	0.496	0.892
PT	CodeBERT	0.370	0.829	0.493	0.880
	GraphCodeBERT	0.333	0.815	0.483	0.886
	PLBART	0.332	0.773	0.403	0.812
	UniXcoder	0.371	0.852	0.517	0.898
	CodeT5	0.310	0.717	0.382	0.768
	CodeT5+	0.297	0.737	0.346	0.779
	Avg.	0.336	0.787	0.437	0.837
PreT	CodeBERT	0.370	0.846	0.510	0.890
	GraphCodeBERT	0.398	0.866	0.526	0.900
	PLBART	0.348	0.791	0.410	0.798
	UniXcoder	0.422	0.858	**0.566**	0.904
	CodeT5	0.299	0.726	0.336	0.778
	CodeT5+	0.277	0.721	0.360	0.782
	Avg.	0.352	0.801	0.451	0.842
ZL	CodeBERT	0.000	0.443	0.148	0.453
	GraphCodeBERT	0.090	0.485	0.141	0.486
	PLBART	0.053	0.476	0.143	0.461
	UniXcoder	0.134	0.498	0.148	0.520
	CodeT5	0.131	0.525	0.134	0.500
	CodeT5+	0.159	0.510	0.156	0.528
	Avg.	0.095	0.489	0.145	0.491
ICL	CodeBERT	0.026	0.514	0.152	0.486
	GraphCodeBERT	0.038	0.483	0.170	0.540
	PLBART	0.064	0.435	0.140	0.480
	UniXcoder	0.125	0.480	0.160	0.561
	CodeT5	0.128	0.501	0.136	0.526
	CodeT5+	0.159	0.493	0.158	0.512
	Avg.	0.090	0.484	0.153	0.517

TABLE III
EVALUATION RESULTS OF CMG ON THE MCMD DATASET. M IS SHORT FOR METEOR. R IS SHORT FOR ROUGE-L. DIGITS IN BOLD DENOTE THE BEST VALUE AMONG ALL RESULTS. TABLE CELLS IN LIGHT GREY BACKGROUND DENOTE THE BEST RESULT AMONG DIFFERENT APPROACHES UNDER THE SAME BACKBONE MODEL

Models		Java			C#			C++			Python			JavaScript			Avg.		
		BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R
CodeReviewer		21.13	12.15	26.69	21.85	13.82	28.84	15.51	10.55	20.50	18.42	**15.72**	**29.95**	20.16	13.57	26.32	19.41	13.16	26.46
CCT5		23.80	14.27	30.52	**23.64**	**15.80**	**29.76**	17.22	12.17	23.22	20.25	14.10	27.55	24.42	16.30	31.78	21.87	14.53	28.57
CodeT5	FMFT	24.34	15.28	32.15	22.11	13.96	28.93	18.78	13.38	25.70	20.55	14.97	28.96	25.03	17.30	32.99	22.16	14.98	29.75
	LoRA	21.65	14.14	29.17	18.39	11.89	24.84	16.97	12.60	23.36	18.76	14.02	26.57	23.48	16.18	31.17	19.85	13.77	27.02
	AT	23.95	15.25	31.60	20.69	13.36	27.49	18.33	13.39	25.31	20.12	14.75	28.44	24.92	17.07	32.75	21.60	14.76	29.12
	PT	11.69	7.89	16.22	10.67	6.71	14.71	9.93	7.53	13.61	10.97	8.17	14.98	13.02	9.28	18.41	11.26	7.92	15.59
	PreT	16.10	10.07	21.18	13.40	8.08	17.92	12.74	9.35	17.27	14.60	11.02	20.70	17.66	12.16	23.73	14.90	10.14	20.16
	ZL	0.79	1.25	1.26	0.84	1.37	1.05	1.14	1.91	1.34	1.22	2.16	1.43	0.86	1.45	1.57	0.97	1.63	1.33
	ICL	0.73	1.20	2.43	0.87	1.03	2.14	0.94	1.14	1.54	1.05	1.04	3.63	0.67	1.20	3.02	0.85	1.12	2.55
CodeT5+	FMFT	**24.82**	**15.87**	**32.88**	22.51	14.37	**29.76**	**19.28**	**14.11**	**26.55**	**21.11**	15.65	29.74	**26.08**	**17.67**	**34.10**	**22.76**	**15.53**	**30.61**
	LoRA	21.74	14.09	29.25	18.40	11.73	25.03	17.05	12.46	23.13	18.60	10.97	26.86	22.88	15.76	30.42	19.73	13.00	26.94
	AT	23.87	15.42	31.88	21.04	13.43	27.83	18.14	13.28	25.15	19.95	15.08	28.44	25.23	17.47	33.53	21.65	14.94	29.37
	PT	8.49	4.51	11.63	7.86	3.79	10.57	8.84	5.94	11.41	10.35	6.68	12.95	10.17	7.47	15.41	9.14	5.68	12.39
	PreT	16.62	9.40	22.17	13.21	8.45	17.93	12.66	7.21	17.72	13.66	6.94	19.53	17.06	9.81	23.28	14.64	8.36	20.13
	ZL	0.99	1.15	0.53	1.12	1.13	0.52	1.28	1.13	0.54	1.29	1.14	0.62	1.24	1.14	0.56	1.18	1.14	0.55
	ICL	0.76	1.05	0.27	1.06	1.05	0.63	1.03	1.05	0.20	1.06	1.07	0.25	1.10	1.05	0.24	1.00	1.05	0.32
Qwen2.5 -Coder	FMFT	13.40	7.57	17.72	12.10	6.97	16.17	10.03	5.01	13.74	9.94	6.54	14.76	13.58	8.48	18.76	11.81	6.91	16.23
	LoRA	12.72	6.52	16.41	11.32	5.05	14.26	9.65	4.93	12.72	9.28	4.78	12.76	12.77	5.80	16.73	11.15	5.14	14.58
	AT	14.63	7.64	19.05	13.75	7.03	17.71	11.51	5.98	15.63	11.27	6.91	16.17	14.49	7.76	19.32	13.13	7.06	17.58
	PT	1.13	3.68	1.52	1.14	3.02	1.41	1.22	3.03	1.47	1.31	3.11	1.77	1.21	2.96	1.38	1.20	3.16	1.51
	PreT	1.12	3.65	1.53	1.13	2.90	1.39	1.22	3.02	1.47	1.31	3.13	1.76	1.21	2.97	1.37	1.20	3.13	1.50
	ZL	1.13	3.68	1.52	1.14	3.02	1.41	1.22	3.03	1.47	1.31	3.11	1.77	1.21	2.96	1.38	1.20	3.16	1.51
	ICL	1.05	3.02	1.45	1.05	3.26	1.70	1.21	2.65	1.42	1.28	3.13	1.93	1.02	3.20	1.61	1.12	3.05	1.62

Finding 1: In the JIT-DP task, all four PEFT methods can surpass FMFT when incorporating extra expert features. Notably, AT and LoRA even obtain improvements of 5.08% and 9.75% in terms of F1 when compared with SOTA baselines.

2) Commit Message Generation: As for the commit message generation task, we employ CodeT5, CodeT5+, and Qwen2.5-Coder as the backbone PLMs to explore the performance of PEFT methods. Considering there are five sub-datasets containing diverse PLs, namely Java, C#, C++, Python, and JavaScript, we fine-tune PLMs for each PL separately and calculate the average performance among the five PLs. The evaluation results are shown in Table III. For CodeReviewer and CCT5, since their performance on Meteor and Rouge-L is not reported in the original papers, we retrained them using their source code and report the full evaluation results.

From Table III, when using CodeT5 and CodeT5+ as backbones, we could observe mild degradation with AT and LoRA compared to FMFT. The degradation is consistent across the five PLs. For example, when applied to CodeT5, AT and LoRA achieve average BLEU scores of 21.60 and 19.85, respectively, both of which are slightly below the FMFT's result of 22.16. However, PT and PreT significantly underperform FMFT in the CMG task, which is in agreement with the findings in previous studies [10], [36]. For the decrements of PEFT methods, we conjecture the reasons mainly contain: (i) In the architecture of CodeT5 and CodeT5+, AT, LoRA, PT, and PreT only modify 4.10%, 0.40%, 0.03%, and 0.41% of the full parameters, which are quite a small portion of parameters. (ii) Compared with classification tasks, e.g., the JIT-DP task, generation tasks, such as the CMG task, carry a high complexity and difficulty. As such, the CMG task needs more parameter tuning for knowledge adaptation. (iii) As for the significant underperformances of PT and PreT, because the two methods only optimize the representation of prompt tokens but ignore other input tokens, and PT only involves parameters in the first layer, we argue that it is not enough to transfer pre-trained knowledge to the CMG task, and thus leads to the degradation. Their underperformance has also been observed in prior studies [36], [73], underscoring both the inherent complexity of dynamic code comprehension tasks and the inadequacy of these methods' limited parameter budgets in capturing the sophisticated knowledge adaptation required by such tasks.

When applying FMFT, AT, and LoRA to Qwen2.5-Coder, their performance is observed to be lower than that achieved with CodeT5 and CodeT5+. Since Qwen2.5-Coder was pre-trained with next token prediction and fill-in-the-middle without any code diff semantic learning, making them less capable of modeling the transformation from code changes to commit messages. In contrast, CodeT5 and CodeT5+ were pre-trained with identifier-aware denoising, simulating code snippet modifications, thereby making them learn to restore masked identifiers or code blocks, which is directly associated with code change scenarios. Nevertheless, we can still observe similar

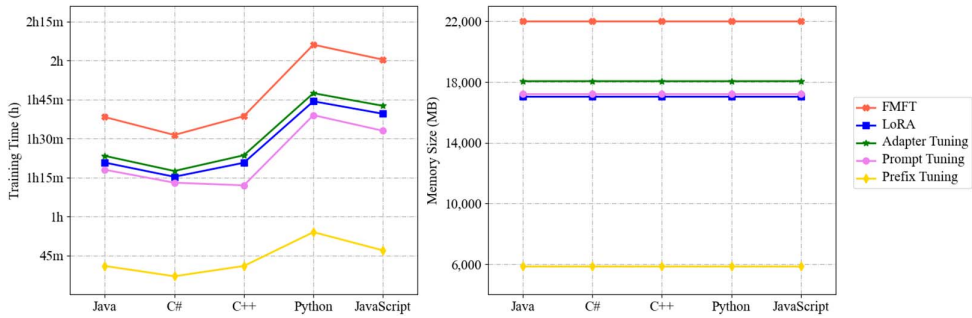


Fig. 4. Comparison of training time and memory size in the CMG task, using CodeT5 as the backbone.

conclusions to those of CodeT5 and CodeT5+, namely, AT and LoRA achieve performance on par with FMFT. For PT and PreT, they perform poorly and fail to generate high-quality commit messages, further revealing their limitations when applied to PLMs.

When compared with CodeReviewer and CCT5, AT and LoRA perform neck-to-neck when using CodeT5 and CodeT5+ as backbones, especially dominating on sub-datasets of Java, C++, and JavaScript. By contrast, PT and PreT perform weakly. It is well known that pre-training on large-scale datasets is not practical for everyone, but AT and LoRA can obtain similar results by modifying only a small portion of the parameters, indicating their practical value. Additionally, in line with the findings from the JIT-DP task, ZL and ICL continue to perform significantly worse in the CMG task, highlighting their limitations without fine-tuning.

Finding 2: In the CMG task, PEFT methods perform lower than or on par (at best) with FMFT. Except for FMFT, AT and LoRA still perform the best among other domain adaptation approaches. Besides, using CodeT5 and CodeT5+ as backbones, AT and LoRA perform neck-to-neck with SOTA baselines.

3) Efficiency: In addition to evaluating the performance, we also evaluate the efficiency of PEFT methods. Taking CodeT5 in the CMG task as a case study, we analyze the training time and memory consumption of FMFT and the four PEFT methods. The results are illustrated in Fig. 4. We report the average training time of each epoch and the occupied memory size in total. The devices we use are elaborated in Section III-E. It can be observed that AT, LoRA, and PT can save approximately 15-30 minutes each epoch, and decrease memory consumption of about 4,000-5,000 MB. PreT can save more training time and computational resources, which are about 1 hour and 16,000 MB of memory size.

Finding 3: When compared with FMFT, PEFT methods consume less training time and memory size, which can save up to 1 hour and 16,000 MB at most in code-change-related tasks.

4) Hyperparameter Analysis: To conduct a thorough analysis of the impact of PEFT-related hyperparameters, specifically,

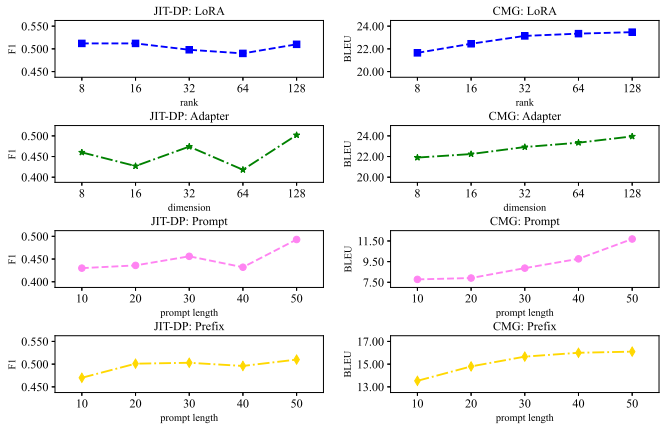


Fig. 5. Hyperparameter analysis of the LoRA rank, the intermediate dimension for AT, and the prompt length for PT and PreT.

the LoRA rank, intermediate dimension of AT, and prompt length for PT and PreT, we adjust these critical hyperparameters within appropriate ranges and observe the resulting performance variations. For the JIT-DP task, we employ CodeBERT as the backbone model under the w/ EF setting, while for the CMG task, we use CodeT5 on the Java dataset. As shown in Fig. 5, the trends in performance with respect to the LoRA rank and the intermediate dimension of AT vary across tasks. In the JIT-DP task, the best performance for LoRA is achieved at a rank of 8. As the rank increases from 8 to 128, performance first declines and then improves. Similarly, AT exhibits fluctuating performance as the dimension changes, achieving its peak at a dimension of 128, consistent with our chosen hyperparameter configuration. In the CMG task, the performance of respective PEFT methods consistently improves as the LoRA rank and the intermediate dimension of AT increase. But it is worth noting that while larger values yield better performance, they also lead to a higher number of trainable parameters. Regarding the prompt length for PT and PreT, within our explored range, performance consistently shows an upward trend as the prompt length increases across both tasks, achieving the best results at a length of 50. The above analysis demonstrates that the hyperparameter settings of diverse PEFT methods are well optimized overall, thereby ensuring the reliability of this work.

TABLE IV
EVALUATION RESULTS OF CMG IN THE CROSS-LINGUAL SCENARIO. CODET5 IS USED AS THE BACKBONE MODEL FOR FMFT, LORA, AT, PT, AND PRET

Models		Java	C#	C++	Python	JavaScript	Avg.
Java	FMFT	-	9.94	10.06	10.95	12.10	10.76
	LoRA	-	**10.52**	**11.00**	**11.70**	**13.30**	**11.63**
	AT	-	10.47	10.69	11.31	12.90	11.34
	PT	-	8.41	8.61	9.84	10.49	9.34
	PreT	-	8.72	9.16	9.97	10.57	9.61
C#	FMFT	10.23	-	9.97	10.68	13.18	11.02
	LoRA	**11.01**	-	**11.04**	**11.57**	13.21	**11.71**
	AT	10.65	-	10.89	11.44	**13.34**	11.58
	PT	9.39	-	9.77	10.54	11.40	10.28
	PreT	9.16	-	9.60	10.65	10.76	10.04
C++	FMFT	10.64	10.42	-	12.08	13.44	11.65
	LoRA	**11.26**	**10.93**	-	**12.64**	**14.20**	**12.26**
	AT	11.00	10.31	-	12.34	13.52	11.79
	PT	8.08	8.87	-	10.02	10.56	9.38
	PreT	8.10	7.91	-	10.20	9.87	9.02
Python	FMFT	9.78	10.14	10.47	-	13.24	10.91
	LoRA	**10.20**	10.54	10.99	-	**13.79**	**11.38**
	AT	10.00	**10.55**	**11.03**	-	13.69	11.32
	PT	8.24	8.42	9.60	-	10.19	9.11
	PreT	9.17	9.30	10.20	-	11.49	10.04
JavaScript	FMFT	9.62	9.90	9.75	10.93	-	10.05
	LoRA	**11.37**	**10.73**	**11.20**	**12.30**	-	**11.40**
	AT	10.65	9.96	10.63	11.88	-	10.78
	PT	8.95	8.56	8.81	9.88	-	9.05
	PreT	9.48	8.93	9.51	10.73	-	9.66

Finding 4: The impact of the LoRA rank and the intermediate dimension of AT varies across tasks, while within our explored range, the performance of PT and PreT consistently exhibits an upward trend as the prompt length increases.

B. RQ2: Performance in the Cross-lingual Scenario

In this section, we investigate the performance of PEFT methods in the cross-lingual scenario, comparing them with FMFT as well. Using CodeT5 as the backbone, we apply these methods to the CMG task as its dataset contains five PLs. In the cross-lingual scenario, PLMs are expected to generate commit messages about the target PL but are fine-tuned in a different source PL. It can be seen as a combination task of monolingual CMG and program translation [74], and is useful when the training data is limited or developers are only familiar with specific PLs. Table IV illustrates the evaluation results, where rows represent the source PLs while columns represent the target PLs. We only report the BLEU metric due to the page limitation.

From Table IV, eliminating the diagonal values, i.e., the source PL and target PL are the same, we could observe consistent improvements over FMFT brought by AT and LoRA in the cross-lingual scenario. Specifically, the average promotion of AT and LoRA can be up to 4.62% and 7.56%, respectively. A potential explanation is that FMFT, after updating all parameters, overfits to the training samples (i.e., the source PL). In contrast, LoRA and AT, which adjust only a small portion of parameters, preserve the multilingual general knowledge acquired during PLMs' pre-training and further learn new PL-invariant knowledge from the training samples, enabling them to apply the above general knowledge to the testing set (i.e., the target PL) and outperform FMFT in cross-lingual scenarios. As for PT and PreT, their average performance is still weaker than FMFT, but the declines are slight compared with the results in RQ1. Because, although they also maintain the previous PL-invariant knowledge from their backbone PLMs, their trainable parameters are extremely limited, making it insufficient to learn PL-invariant features from the training samples (i.e., the source PL). Therefore, PreT and PT do not reach LoRA and AT's performance in the cross-lingual scenario, but still manifest a comparable performance to FMFT. Given the much lower computational resources required and the comparable or even better performance of PEFT methods, we strongly recommend using PEFT techniques for cross-lingual code change learning. The above findings bring unique insights. PEFT is particularly suitable for cross-lingual scenarios because it updates only a small portion of parameters, thereby substantially mitigating catastrophic forgetting of previously acquired general knowledge in backbone PLMs. Besides, partially trainable parameters also leave PEFT methods room to further acquire PL-invariant knowledge for follow-up transfer to unseen PLs, while this process remains effective for code change learning.

Finding 5: PEFT techniques perform either better or comparable to FMFT in cross-lingual scenarios. Considering their much lower computational resource consumption, PEFT is highly recommended.

C. RQ3: Performance in the Low-Resource Scenario

In this section, we investigate how well the PEFT methods perform in the low-resource scenario. Considering the disparity in dataset sizes between the two tasks, we randomly sample 1,000, 5,000, and 10,000 training instances for each PL in the CMG task, and 100, 500, and 1,000, 2500, and 5000 samples for JIT-DP. We repeat experiments three times with different random seeds to mitigate the bias in data sampling on the results. In addition, we employ the Wilcoxon Signed-Rank Test (WSRT) [75] with a confidence level of 95% to conduct pairwise comparisons between FMFT and PEFT methods. The evaluation results are presented in Tables V and VI, respectively. Owing to the page limit, we only report standard deviations under the **Avg.** column of Table V. In contrast, in Table VI, standard deviations are listed across all columns. For PEFT methods that outperform FMFT with a statistically significant difference on certain metrics, we highlight them in red under those metrics. In contrast, if FMFT statistically significantly outperforms PEFT methods, we highlight them in green accordingly. For the remaining ones without statistically significant differences, we leave them intact (i.e., white background).

As shown in Table V, it can be observed that when reducing the training samples to 1,000, 5,000, and 10,000 for

TABLE V
EVALUATION RESULTS OF CMG IN THE LOW-RESOURCE SCENARIO. M IS SHORT FOR METEOR. R IS SHORT FOR ROUGE-L. CODET5 IS USED AS THE BACKBONE MODEL FOR FMFT, LORA, AT, PT, AND PRET

Samples	Models	Java			C#			C++			Python			JavaScript			Avg.		
		BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R	BLEU	M	R
1,000	FMFT	**10.32**	**7.19**	**14.23**	**9.81**	**6.33**	**13.56**	**9.04**	7.02	**12.42**	**10.73**	**7.99**	**14.57**	10.71	**8.25**	15.57	**10.12**$_{\pm 0.79}$	**7.36**$_{\pm 0.19}$	**14.07**$_{\pm 0.38}$
	LoRA	9.47	6.88	13.34	9.22	5.59	12.45	8.93	6.89	12.11	10.31	7.81	13.53	**11.09**	8.07	**16.08**	9.80$_{\pm 0.24}$	7.05$_{\pm 0.10}$	13.50$_{\pm 0.17}$
	AT	10.08	6.80	13.90	9.43	6.14	13.19	8.64	**7.15**	11.99	9.19	6.88	12.25	9.78	7.93	14.77	9.42$_{\pm 0.12}$	6.98$_{\pm 0.20}$	13.22$_{\pm 0.31}$
	PT	5.97	3.53	7.92	6.80	3.86	8.99	7.65	5.44	10.27	8.17	5.52	10.52	6.98	4.59	9.61	7.11$_{\pm 0.17}$	4.59$_{\pm 0.24}$	9.46$_{\pm 0.42}$
	PreT	4.95	2.88	7.11	5.82	3.09	7.71	6.84	4.75	8.49	6.68	4.47	7.98	6.29	3.98	8.95	6.11$_{\pm 0.17}$	3.83$_{\pm 0.06}$	8.05$_{\pm 0.22}$
5,000	FMFT	14.32	**9.57**	19.44	12.84	7.59	17.03	**11.84**	**8.85**	16.03	12.77	**9.89**	**18.16**	**16.03**	**11.08**	**21.82**	**13.56**$_{\pm 0.34}$	9.39$_{\pm 0.42}$	18.49$_{\pm 0.55}$
	LoRA	14.22	9.51	19.27	12.36	7.76	16.80	11.16	8.40	15.48	12.72	9.80	17.73	15.25	10.75	20.96	13.14$_{\pm 0.15}$	9.25$_{\pm 0.17}$	18.05$_{\pm 0.24}$
	AT	**14.33**	9.48	**19.50**	**12.90**	**8.10**	**17.59**	11.47	8.82	**16.04**	**13.00**	9.88	18.15	15.50	11.02	21.50	13.44$_{\pm 0.28}$	**9.46**$_{\pm 0.28}$	**18.56**$_{\pm 0.35}$
	PT	7.35	5.06	10.50	7.24	4.47	9.94	8.13	6.35	11.28	9.10	6.80	12.11	9.03	6.56	12.90	8.17$_{\pm 0.22}$	5.85$_{\pm 0.20}$	11.35$_{\pm 0.36}$
	PreT	6.83	4.91	9.61	6.82	4.42	9.30	7.39	5.57	9.65	8.07	5.83	10.36	8.61	6.19	12.51	7.54$_{\pm 0.13}$	5.38$_{\pm 0.17}$	10.29$_{\pm 0.09}$
10,000	FMFT	**17.19**	**11.25**	**23.15**	14.15	8.63	**19.22**	**13.05**	**9.65**	**18.19**	14.76	11.28	21.22	**18.05**	**12.75**	**24.78**	**15.44**$_{\pm 0.02}$	**10.71**$_{\pm 0.04}$	**21.31**$_{\pm 0.13}$
	LoRA	16.29	10.86	22.01	13.80	9.11	18.82	12.73	9.63	17.56	14.28	11.07	20.43	17.51	12.44	23.95	14.92$_{\pm 0.04}$	10.62$_{\pm 0.05}$	20.55$_{\pm 0.08}$
	AT	16.77	10.85	22.50	**14.28**	**9.13**	19.07	12.70	9.55	17.31	**14.91**	**11.40**	**21.86**	17.66	12.29	24.08	15.26$_{\pm 0.12}$	10.64$_{\pm 0.18}$	20.96$_{\pm 0.05}$
	PT	7.62	4.82	10.47	7.42	4.81	10.41	8.16	6.74	11.86	9.59	6.95	12.55	9.56	7.07	13.98	8.47$_{\pm 0.38}$	6.08$_{\pm 0.11}$	11.85$_{\pm 0.42}$
	PreT	9.44	6.61	13.09	8.83	5.69	12.17	8.07	6.17	11.03	9.35	6.91	12.56	10.97	7.38	15.17	9.33$_{\pm 0.28}$	6.55$_{\pm 0.19}$	12.80$_{\pm 0.38}$

*A light green background indicates that FMFT outperforms the corresponding PEFT methods with statistical significance, while the remaining ones indicate no statistically significant differences.

TABLE VI
EVALUATION RESULTS OF JIT-DP IN THE LOW-RESOURCE SCENARIO. CODEBERT IS USED AS THE BACKBONE MODEL

Samples	Models	w/o EF		w/ EF	
		F1	AUC	F1	AUC
100	FMFT	0.123$_{\pm 0.118}$	0.691$_{\pm 0.073}$	0.117$_{\pm 0.131}$	0.681$_{\pm 0.077}$
	LoRA	0.015$_{\pm 0.014}$	0.671$_{\pm 0.010}$	0.079$_{\pm 0.076}$	0.667$_{\pm 0.080}$
	AT	**0.138**$_{\pm 0.124}$	**0.708**$_{\pm 0.053}$	0.195$_{\pm 0.152}$	**0.708**$_{\pm 0.049}$
	PT	0.022$_{\pm 0.015}$	0.672$_{\pm 0.002}$	0.203$_{\pm 0.110}$	0.664$_{\pm 0.073}$
	PreT	0.130$_{\pm 0.115}$	0.648$_{\pm 0.047}$	**0.209**$_{\pm 0.078}$	0.668$_{\pm 0.067}$
500	FMFT	0.269$_{\pm 0.108}$	0.795$_{\pm 0.014}$	0.296$_{\pm 0.073}$	0.795$_{\pm 0.023}$
	LoRA	0.271$_{\pm 0.076}$	0.797$_{\pm 0.016}$	0.313$_{\pm 0.104}$	**0.818**$_{\pm 0.021}$
	AT	**0.301**$_{\pm 0.047}$	**0.801**$_{\pm 0.015}$	0.308$_{\pm 0.073}$	0.807$_{\pm 0.021}$
	PT	0.117$_{\pm 0.090}$	0.751$_{\pm 0.018}$	0.307$_{\pm 0.058}$	0.779$_{\pm 0.030}$
	PreT	0.254$_{\pm 0.092}$	0.777$_{\pm 0.033}$	**0.326**$_{\pm 0.033}$	0.796$_{\pm 0.035}$
1,000	FMFT	0.284$_{\pm 0.054}$	**0.818**$_{\pm 0.019}$	0.345$_{\pm 0.022}$	0.813$_{\pm 0.011}$
	LoRA	0.281$_{\pm 0.062}$	0.810$_{\pm 0.010}$	**0.358**$_{\pm 0.049}$	**0.837**$_{\pm 0.009}$
	AT	0.305$_{\pm 0.069}$	0.809$_{\pm 0.032}$	0.259$_{\pm 0.080}$	0.828$_{\pm 0.011}$
	PT	0.110$_{\pm 0.065}$	0.764$_{\pm 0.021}$	0.292$_{\pm 0.077}$	0.804$_{\pm 0.028}$
	PreT	**0.326**$_{\pm 0.046}$	0.814$_{\pm 0.018}$	0.356$_{\pm 0.086}$	0.831$_{\pm 0.029}$
2,500	FMFT	0.279$_{\pm 0.038}$	0.820$_{\pm 0.010}$	0.307$_{\pm 0.055}$	0.831$_{\pm 0.021}$
	LoRA	**0.290**$_{\pm 0.035}$	**0.826**$_{\pm 0.005}$	**0.384**$_{\pm 0.012}$	**0.857**$_{\pm 0.015}$
	AT	0.281$_{\pm 0.056}$	0.817$_{\pm 0.010}$	0.266$_{\pm 0.028}$	0.827$_{\pm 0.004}$
	PT	0.094$_{\pm 0.012}$	0.765$_{\pm 0.022}$	0.289$_{\pm 0.022}$	0.801$_{\pm 0.002}$
	PreT	0.268$_{\pm 0.039}$	0.800$_{\pm 0.006}$	0.255$_{\pm 0.023}$	0.812$_{\pm 0.026}$
5,000	FMFT	0.311$_{\pm 0.010}$	0.843$_{\pm 0.003}$	0.349$_{\pm 0.046}$	0.850$_{\pm 0.010}$
	LoRA	**0.323**$_{\pm 0.024}$	**0.847**$_{\pm 0.004}$	**0.463**$_{\pm 0.020}$	**0.873**$_{\pm 0.009}$
	AT	0.297$_{\pm 0.029}$	0.828$_{\pm 0.006}$	0.291$_{\pm 0.010}$	0.841$_{\pm 0.012}$
	PT	0.105$_{\pm 0.026}$	0.814$_{\pm 0.007}$	0.396$_{\pm 0.030}$	0.864$_{\pm 0.003}$
	PreT	**0.323**$_{\pm 0.037}$	0.833$_{\pm 0.002}$	0.390$_{\pm 0.029}$	0.853$_{\pm 0.008}$

*A light green background indicates that FMFT outperforms the corresponding PEFT method with statistical significance, while a light red background indicates the reverse, i.e., the PEFT method significantly outperforms FMFT. The remaining ones indicate no statistically significant differences.

each PL ($< 1/100$, $< 1/20$, and $< 1/10$ of the full-data setting), AT and LoRA still slightly lag behind FMFT on the CMG task. Moreover, the WSRT shows that the advantage of FMFT is statistically significant across almost all metrics. However, compared with the full-data setting, where FMFT always outperforms AT and LoRA (see RQ1), they exhibit relative superiority in the low-resource setting, with distinct strengths across different PLs. In particular, in the 5,000-sample setting, AT, on average, outperforms FMFT on the Meteor and Rouge-L metrics, although the WSRT does not demonstrate a statistically significant difference. As such, we conclude that AT and LoRA can achieve comparable or slightly inferior performance in low-resource settings. We speculate that CMG is a challenging code-change task that requires more parameter tuning for adaptation. Even though AT and LoRA inherently exhibit superiority in low-resource settings [17], [36], they still cannot achieve a better understanding of code changes with very limited training samples in such generative tasks. As for PT and PreT, they still show poor performance compared with FMFT, and the gaps narrow as the number of training samples decreases. Given their efficiency during tuning, they can be viable alternatives when training samples are very limited. We also compare the above methods with CodeT5 without a fine-tuned version, i.e., in ZL and ICL settings. Since the testing sets are fixed to the same as those in RQ1, and we only change the random seeds to sample from the training set, the results for the ZL and ICL settings can be found in Table III. We find that both settings perform worse than PEFT/FMFT methods and lag behind them significantly, as the parameter size of CodeT5 is limited and lacks emerging abilities [76].

For the JIT-DP task, we define five settings with varying training samples, the largest of which (5,000 samples) is still less than one-third of the full-data setting. To reduce the influence of data sampling variability, we perform three independent experiments with distinct random seeds and report the average results. From Table VI, we can observe that most PEFT methods achieve comparable performance against FMFT under the w/o EF setting. PT performs the worst in most low-resource settings. When incorporating expert features (i.e., the

w/ EF setting), both FMFT and PEFT show improvements, with PEFT's gain being more substantial, and many PEFT methods start to surpass FMFT with a statistically significant margin (highlighted in red). This phenomenon is the same as that in RQ1, shown in Table II. In addition, we find that these methods perform inconsistently when the training sample size is small. For example, LoRA, AT, PreT, and FMFT all have a chance to achieve the best performance on certain metrics when the training sample size is limited to 100, 500, or 1,000. Besides, their standard deviations are relatively larger than those in 2,500 or 5,000 training samples, suggesting that the training is possibly insufficient and their performance is highly affected by data sampling. When it comes to 2,500 and 5,000 training samples, we find that LoRA consistently maintains its advantage over diverse counterparts, including FMFT. To be specific, LoRA surpasses them on average by 55.06%–55.97% in F1 and 2.13%–3.26% in AUC with the w/o EF setting. As for the w/ EF setting, LoRA still maintains its superiority by 31.85%–38.23% in F1 and 2.47%–4.82% in AUC on average. This indicates that LoRA indeed outperforms other PEFT methods and FMFT on the JIT-DP task, provided it is sufficiently trained. Similarly, the ZL and ICL settings still perform the worst and lag behind the above methods a lot, as shown in Table II.

Finding 6: In the CMG task, almost all PEFT methods fall short of FMFT in the low-resource scenario. In contrast, for the JIT-DP task in the w/ EF setting, PEFT methods can overall achieve superior or at least comparable performance to FMFT when training data is limited.

D. RQ4: Probing Tasks

In this section, we utilize three probing tasks, namely TYP, CCM, and LTP tasks, to explore what code properties are encoded in PLMs, thereby making an explanation for the above experimental results. TYP task relates to static semantic information, while CCM and LTP tasks are associated with dynamic semantic information from global and local perspectives. We reused CodeBERT and CodeT5 trained with FMFT, four PEFT methods, and ZL settings for JIT-DP and CMG tasks in RQ1 for layer-wise probing experiments. Considering the PLM's parameters in ZL and ICL are the same, as they are all fixed without training, we only consider the former setting for probing experiments. For each probing task, after tuning on their respective training set, we extract the encoded features from each transformer layer and then feed them into a simple linear classifier to predict their categories [5] on their testing sets. The classifier has no hidden units, so the performance of probing tasks prohibitively depends on the feature vectors. The probing results are illustrated in Fig. 6, where the dashed lines indicate the average performance of all layers.

From the first row of Fig. 6, we could derive several insightful findings for the JIT-DP task: (i) The average performances of PEFT methods can always outperform FMFT in all three probing tasks, except for AT and PT that slightly underperform in the LTP task. (ii) Comparing with the advantages obtained by the four PEFT methods in the TYP task, the promotions on CCM and LTP are not dramatic. (iii) The ZL approach achieves the best performance in TYP, but it performs worse than FMFT and PEFT methods on CCM and LTP in most cases. The first finding is expected and explains that the four PEFT methods learn more about code-change-related semantics from both global and local perspectives while keeping the original understanding ability of static code semantics. The second finding indicates that learning dynamic code semantics is more difficult than static semantics. Considering that code-change-related tasks are more complex, the result is reasonable. The third finding is also expected. Since TYP is a probing task related to static code semantics, fine-tuning on code-change-related downstream tasks tends to reduce the model's capacity for understanding static code properties. In contrast, CCM and LTP probe dynamic code semantics, which are less directly captured by pre-training. As a result, the ZL approach performs poorly in these aspects, while FMFT and PEFT methods demonstrate superior adaptability in most cases. It can be concluded that the probing results are overall consistent with previous findings and explain the efficacy of PEFT methods.

For the CMG task, as shown in the second row of Fig. 6, we found that no one approach can dominate all three probing tasks. The ZL setting still performs best in the TYP task, as PLMs' parameters were not adapted to either JIT-DP or CMG task under this approach, thereby keeping its original performance on static code understanding. As for probing tasks of dynamic perspective, AT dominates on the CCM task, while FMFT excels on the LTP task. Hence, it is hard to say which method leads PLMs to learn more about dynamic code semantics during adaptation. These probing results also reflect the experimental evidence in RQ1 that, although AT and LoRA underperform FMFT slightly, their performance gaps are not significant. However, as for PreT and PT, they always perform the worst in both RQ1 and probing experiments on average. Although the average performances of FMFT and PEFT methods are similar, their layer-wise probing results show distinctive advantages. For example, in the CCM task, AT and LoRA perform much better than FMFT in the top layers (layers 6-12) but significantly decrease in the bottom layers (layers 1-5). This finding enlightens us to combine the layer-wise advantages of different tuning methods to further boost domain adaptation performance from a fine-grained perspective in the following sections.

Finding 7: Probing tasks show that PEFT methods can benefit the understanding of dynamic code semantics in the JIT-DP task. In the CMG task, the dynamic understanding gap between AT, LoRA, and FMFT is not evident on average. Nonetheless, PEFT methods carry distinctive advantages against FMFT in certain layers, making their combination in domain adaptation a reasonable attempt.

E. RQ5: Self-Adaptive Efficient Layer-Specific Tuning

In RQ1, the evaluation results reveal that although efficient, PEFT methods perform either comparable or worse than FMFT

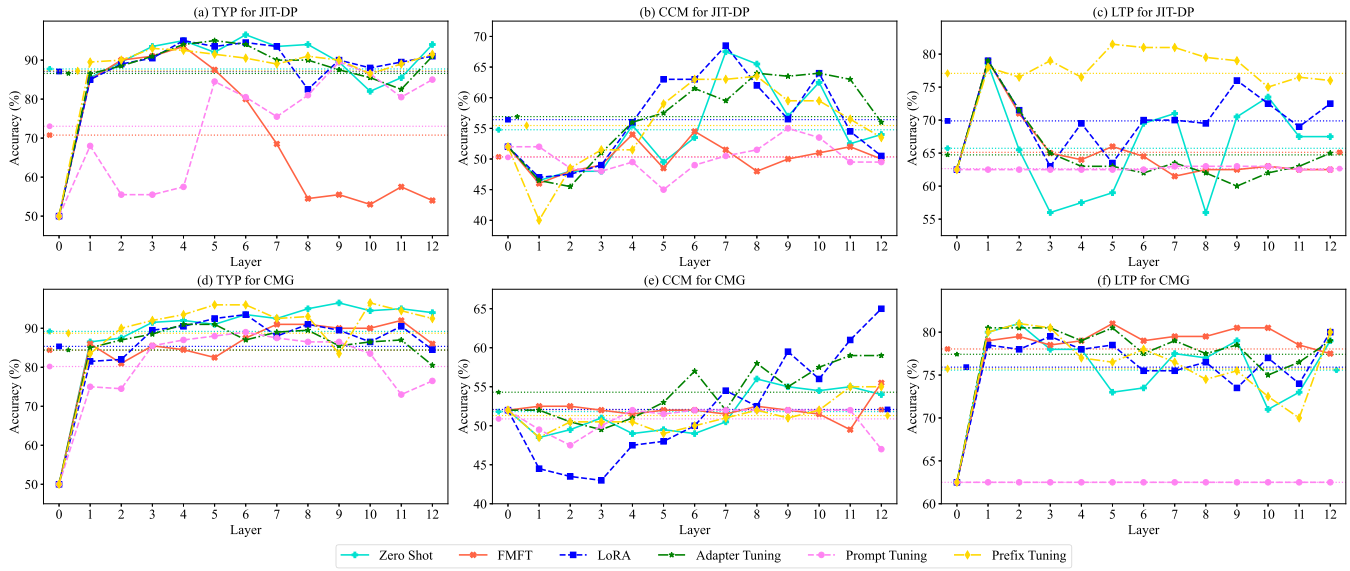


Fig. 6. Accuracy of probing tasks by layers.

in the CMG task. Inspired by the layer-wise probing results in RQ4, we attempt to combine the distinctive advantages of PEFT and FMFT by extending PLMs' parameters of certain layers to be trainable in PEFT methods, consequently achieving both high performance and efficiency. To that end, we propose **Pasta$_K$**, which stands for self-adaPtive efficient layer-specific tuning for pre-trained models in code change learning. K is a hyperparameter that represents the number of FMFT layers. Given a specific PEFT method, Pasta$_K$ first determines the location of the PEFT layers and FMFT layers by the layer-wise probing results of either CCM or LTP. For example, to combine LoRA and FMFT in the CMG task, we use CCM (see Fig. 6(e)) for location determination, as it presents more obvious layer-wise pros and cons of LoRA and FMFT than LTP (see Fig. 6(f)), where LoRA performs better at top layers (layer# $\geq$ 5), while FMFT excels at the bottom layers (layer# $\leq$ 4). Given the pre-defined FMFT layer number (K), Pasta$_K$ assigns the bottom K layers to be trainable with FMFT, while tuning the remaining layers with LoRA. As for other PEFT methods, we follow the above principle to implement Pasta$_K$. Since the layer assignments for FMFT and PEFT in Pasta$_K$ are determined from probing results, which can be automatically conducted in practice, the system is self-adaptive. In experiments, since the backbone models we used have 12 layers in total, we uniformly set K to 0, 3, 6, 9, and 12 for evaluation, where 0 and 12 denote **Pasta$_K$** degenerate to its corresponding PEFT and FMFT, respectively, thereby using K to simulate the constraint on the number of full-parameter trainable layers, i.e., the resource constraint. Our experiments in the CMG task are conducted on the Java dataset. The experimental results are shown in Table VII.

For PT, as its poor performance in LTP (see Fig. 6(f)), we can only summarize its layer-wise advantages from CCM (see Fig. 6(e)). As can be seen, PT generally performs neck-to-neck

TABLE VII
EVALUATION RESULTS OF PASTA$_K$ IN THE CMG TASK. WE USE THE JAVA DATASET FOR EVALUATION. THE IMPROVING RATIOS OF PASTA$_K$ ON PERFORMANCE ARE COMPARED WITH CORRESPONDING PEFT METHODS, WHILE ON CONSUMPTION ARE COMPARED WITH FMFT

Models		Performance			Consumption	
		BLEU	M	R	Training Time	Memory Size
FMFT		24.34	15.28	32.15	1h39m	22,005M
LoRA	PEFT	21.65	14.14	29.17	1h20m	17,027M
	$Pasta_3$	23.11 ↑6.74%	14.94 ↑5.66%	30.89 ↑5.90%	**1h25m** ↓14.14%	**17,961M** ↓18.38%
	$Pasta_6$	24.09 ↑11.27%	15.50 ↑9.62%	32.14 ↑10.18%	1h31m ↓8.08%	19,345M ↓12.09%
	$Pasta_9$	**24.47** ↑13.03%	**15.73** ↑11.24%	**32.57** ↑11.66%	1h38m ↓1.01%	20,741M ↓5.74%
AT	PEFT	23.95	15.25	31.60	1h23m	18,049M
	$Pasta_3$	23.79 ↓0.67%	15.46 ↑1.38%	31.90 ↑0.95%	**1h26m** ↓13.13%	**18,771M** ↓14.70%
	$Pasta_6$	24.19 ↑1.00%	15.54 ↑1.90%	32.10 ↑1.58%	1h31m ↓8.08%	19,959M ↓9.30%
	$Pasta_9$	**24.49** ↑2.25%	**15.69** ↑2.89%	**32.41** ↑2.56%	1h35m ↓4.04%	21,151M ↓3.88%
PT	PEFT	11.69	7.89	16.22	1h18m	17,221M
	$Pasta_3$	23.22 ↑98.63%	15.01 ↑90.24%	31.02 ↑91.25%	**1h27m** ↓12.12%	**18,765M** ↓14.72%
	$Pasta_6$	**24.37** ↑108.47%	**15.73** ↑99.37%	**32.49** ↑100.31%	1h31m ↓8.08%	20,247M ↓7.99%
	$Pasta_9$	23.95 ↑104.88%	15.44 ↑95.69%	31.94 ↑96.92%	1h36m ↓3.03%	21,771M ↓1.06%
PreT	PEFT	16.10	10.07	21.18	41m	5,867M
	$Pasta_3$	23.33 ↑44.91%	14.81 ↑47.07%	30.84 ↑45.61%	**57m** ↓42.42%	**10,293M** ↓53.22%
	$Pasta_6$	24.42 ↑51.68%	15.63 ↑55.21%	32.27 ↑52.36%	1h05m ↓34.34%	13,869M ↓36.97%
	$Pasta_9$	**24.70** ↑53.41%	**15.77** ↑56.60%	**32.75** ↑54.63%	1h13m ↓26.26%	17,461M ↓20.65%

$^{\Phi}$ FMFT equals $Pasta_{12}$ while PEFT equals $Pasta_0$.

TABLE VIII
EVALUATION RESULTS OF PASTA$_K$ IN THE JIT-DP TASK

Models		Performance		Consumption	
		F1	AUC	Training Time	Memory Size
FMFT		0.419	0.867	6m02s	23,921M
LoRA	PEFT	**0.512**	**0.899**	4m46s	19,284M
	$Pasta_3$	0.415	0.885	5m05s	20,276M
	$Pasta_6$	0.448	0.880	5m23s	21,296M
	$Pasta_9$	0.462	0.882	5m41s	22,662M
AT	PEFT	**0.502**	**0.894**	5m05s	20,064M
	$Pasta_3$	0.415	0.890	5m23s	21,038M
	$Pasta_6$	0.386	0.885	5m35s	22,058M
	$Pasta_9$	0.398	0.887	6m00s	23,846M
PT	PEFT	**0.493**	**0.880**	4m29s	17,902M
	$Pasta_3$	0.432	0.877	4m47s	18,862M
	$Pasta_6$	0.421	0.866	5m05s	20,254M
	$Pasta_9$	0.422	0.874	5m24s	21,646M
PreT	PEFT	**0.510**	**0.890**	4m03s	17,054M
	$Pasta_3$	0.429	0.872	4m21s	17,824M
	$Pasta_6$	0.463	0.886	4m38s	18,510M
	$Pasta_9$	0.414	0.887	4m55s	20,022M

$^{\Phi}$FMFT equals $Pasta_{12}$ while PEFT equals $Pasta_0$.

with FMFT at the top layers (layer# ≥ 4), while underperforming FMFT at the bottom layers (layer# < 4). Thus, Pasta$_K$ extends the bottom K layers to be trainable with FMFT to improve the ability of domain adaptation. Results are shown in Table VII, Pasta$_K$ significantly improves the performance of PT, achieving a maximum enhancement of 108.47%, 99.37%, and 100.31% in terms of BLEU, Meteor, and Rouge-L, respectively, and also surpassing FMFT, when K is set to 6. As for the consumption of computational resources, although there are increases compared with the vanilla PT, the costs of Pasta$_K$ are significantly lower than FMFT. The reduction can reach up to 12.12% and 14.72% in terms of the consumption of training time and memory size, showing the effectiveness and efficiency of Pasta$_K$.

For PreT, Fig. 6(f) demonstrates its pros and cons more apparent, where FMFT significantly surpasses PreT in the top 9 layers, indicating that Pasta$_K$ should switch the top K layers to be trainable while keeping others frozen, showing the opposite phenomenon of PT. Table VII shows that Pasta$_K$ can enhance the performance of PreT by 53.41%, 56.60%, and 54.63% at most in terms of BLEU, Meteor, and Rouge-L, respectively, even exceeding the performance of FMFT by 1.48%, 3.21%, and 1.87% in terms of each metric in order, when K is set to 9. With comparable or better results, the consumption of Pasta$_K$ in training time and memory size is quite fewer than FMFT. For example, Pasta$_3$ can save 42.42% of training time and 53.22% of computational memory, showing better practical values.

For AT and LoRA, they show a similar trend with PT in Fig. 6(e), performing worse in the bottom layers while better in the top layers. It enlightens Pasta$_K$ to place FMFT layers at the bottom but PEFT layers on the top. From Table VII, we can observe that the maximum improvement is 2.25% and 13.03% in terms of BLEU for AT and LoRA, respectively. The best performance of Pasta$_K$ also surpasses FMFT. Meanwhile, the consumption of Pasta$_K$ is still consistently lower than that of FMFT, showing that Pasta$_K$ can achieve better performance with higher efficiency.

Apart from the experiments on the CMG task, we also conduct extra experiments on the JIT-DP task to explore the performance of Pasta$_K$, where the evaluation results are shown in Table VIII. We still use the setting with expert features and CodeBERT as the backbone model, following the guidance from probing results. Similar to the strategy used in the CMG task, Pasta$_K$ first allocates PEFT and FMFT layers based on layer-wise probing results. As shown in Fig. 6(b), AT, LoRA, and PreT perform either much better or comparable to FMFT, excelling on upper layers (layer# > 4). However, PT performs neck-and-neck with FMFT on the CCM task. As for the LTP task, shown in Fig. 6(c), PreT shows advantages against FMFT from layer# 2 to # 12, while neck-to-neck for the rest of the layers. LoRA generally outperforms FMFT from layer# 4 to # 12 and ties on the remaining layers. AT performs neck-to-neck against FMFT across all layers. PT underperforms FMFT in the bottom layers (layer# < 7), while showing a comparable trend in the remaining layers. Consequently, results from both probing tasks direct us to first extend the bottom layers with FMFT, then gradually unfreeze the upper layers. Similarly, K is still set to 0, 3, 6, 9, and 12 for evaluation, where 0 and 12 represent Pasta$_0$ and Pasta$_{12}$ respectively.

As shown in Table VIII, PEFT (i.e., Pasta$_0$) always performs the best, even outperforming FMFT (i.e., Pasta$_{12}$). Our probing results explain this phenomenon: for AT, LoRA, and PreT, FMFT does not exhibit superior capability for capturing dynamic code semantics at either the top or bottom layers. Consequently, applying PEFT to all layers is optimal, leading Pasta$_K$ to degenerate into pure PEFT, i.e., Pasta$_0$. This does not imply that Pasta$_K$ has failed. Instead, it means that Pasta$_K$ is explainable and strictly follows our proposed layer-wise tuning rules, thus it is still working on the JIT-DP task. It should be noted that Pasta$_K$ is not a method that necessarily combines FMFT and PEFT layers. Instead, it adaptively selects between them based on their performance on probing tasks.

An exception is Pasta$_K$ with PT. Theoretically, Pasta$_K$ should have surpassed PT with the bottom K layers becoming FMFT. But the result is not the case. A potential explanation is that the probing results in CCM are somewhat contradictory to those in LTP when comparing PT and FMFT. For example, in CCM, FMFT's main advantage is in layers#4–6, while in LTP, its advantage is concentrated below layer#3. Owing to this reason, Pasta$_K$ does not achieve the expected performance gain.

Finding 8: By analyzing the layer-wise probing results of CCM and LTP, our proposed Pasta$_K$ can significantly improve the performance of PEFT methods in the CMG task, even surpassing FMFT. Meanwhile, the consumption of Pasta$_K$ in training time and memory size is consistently lower than FMFT, showing the practical values of Pasta$_K$. In the JIT-DP task, the performance of Pasta$_K$ also aligns well with the outcomes of our probing experiments overall, underscoring its rationality and interpretability.

V. Implications

This paper presents the first empirical study that investigates the performance of PEFT on code-change-related tasks and proposes a brand-new, effective, and efficient domain adaptation method, namely Pasta$_K$. In this section, we discuss the implications of this work from the perspectives of developers and researchers.

Implications for developers. This study indicates that PEFT can outperform FMFT on the JIT-DP task, as well as AT and LoRA can achieve comparable results on the CMG task with less training time and memory consumption. The results enlighten developers that can leverage PEFT to replace the original FMFT methods in such code-change-related classification tasks, and for generation tasks, when the computational resources are limited, PEFT can be an effective alternative. In addition, the superiority that PEFT shows in the cross-lingual and low-resource scenarios indicates that PEFT can be used in practice when meeting the data scarcity problem. Different from previous studies that show consistent improvements brought by PEFT in code search, code clone, and code summarization tasks [16], [17], our work guides when and how to apply PEFT to code-change-related tasks. In the end, combining PEFT and FMFT, we propose Pasta$_K$, providing practitioners with a brand-new domain adaptation method with high efficacy and efficiency simultaneously for code change learning. With limited training resources, the new proposed method gives developers more flexibility to strike a balance between efficiency and performance, depending on the actual needs.

Implications for researchers. As the first empirical study to explore the performance of PEFT in code change learning, this work demonstrates that although there is an inconsistency between the pre-training tasks used by PLMs and the code-change-related tasks, PEFT methods can still perform better or comparably to FMFT in certain adaptation cases, such as cross-lingual and low-resource scenarios. Moreover, it can be seen that in the JIT-DP task, the performance of PEFT increases when adding the expert features, demonstrating that PLMs with PEFT methods are more skilled in learning straightforward semantics, owing to their limited parameter tuning. This finding deserves to be investigated in other tasks to pursue high performance. In addition, we design two code-change-related probing tasks for investigating the encoded dynamic code semantics from global and local perspectives. However, probing tasks for code-change learning from other angles are also necessary, such as syntax and lexical. Thus, we call for more effective probing tasks related to the code change. Furthermore, we propose a new approach named Pasta$_K$ to explore domain adaptation techniques in a fine-grained manner. As the first attempt, this work combines FMFT and PEFT by their layer-wise probing results and testifies that a balance between efficiency and performance is achievable. To this end, this work can be seen as a footstone, and in the future, whether the combination can be fully automated or at a neuron level is still worth exploring.

VI. Threats to Validity

In this section, we identify threats to the validity of this work from two aspects.

External validity relates to the generalizability of our findings. In this paper, we conduct experiments on two widely studied code-change-related tasks, including JIT-DP and CMG. Although there are other code-change-related tasks, such as patch identification [77] and comment updating [78], JIT-DP, a classification task, and CMG, a generation task, are representative and cover two main categories. In the future, we can include more code-change-related tasks for assessment. Moreover, due to computational resource constraints, although we did not exhaust all PLMs for experiments, we selected seven representative and widely studied code PLMs with diverse architectures and families. Thus, we believe the above threat is minimal.

Internal validity relates to the aspects that could impact the experimental results. A main threat to the internal validity lies in the hyperparameter settings, especially for those PEFT methods' hyperparameters. In this study, we implement the standard architecture of AT and LoRA and follow their default hyperparameter settings. We conduct specific hyperparameter analysis to explore their impact across both tasks. For the PT and PreT, we also search for a good prompt length among many alternatives. As finding optimal hyperparameters is always a difficult task, it needs to be explored persistently.

VII. Related Work

A. Code Change Learning

Code changes are closely associated with software development and exist extensively. It appears when adding new features, fixing bugs, or refactoring current code [78], [79]. Along with code changes, developers have to undertake considerable workloads for tasks like writing high-quality commit messages [80] and identifying whether software changes are defect-inducing [59]. Therefore, techniques aiming at automating such code-change-related tasks come out and show their effectiveness in enhancing development efficiency. Most of the works follow the representation learning approach, which converts code changes into feature vectors primarily, and then retrieves them in the feature space to obtain expected results.

Concentrating on concrete downstream tasks, some studies propose task-specific methods to learn code change representations. For commit message generation, CoDiSum [81] extracts code structure and code semantics by separating bidirectional GRU and aligns the two parts by an attention layer. FIRA [67] describes code change operations in fine-grained graphs and utilizes a graph-neural-network-based encoder to learn representations. RACE [63], based on the Transformer architecture, retrieves instructive code changes firstly by the cosine similarity of representation vectors, using them to guide the generation of commit messages. For just-in-time defect prediction, DeepJIT [82] leverages two discrete CNNs to extract features from code changes and corresponding commit messages, and employs a fully connected network for feature fusion. JITLine

[45] extracts bag-of-tokens features and conducts five well-known classification techniques, like Support Vector Machine (SVM), to build commit-level prediction models. JIT-Fine [47] combines 14 change-level expert features [59] with extracted semantic features, obtaining integrated representations.

There are also some works focusing on general-purpose code change representations. CC2Vec [46] inputs the removed code and added code separately to a hierarchical attention network, extracts their features, and uses multiple comparison functions to fuse the two parts. CCRep [83] proposes a novel mechanism called query back, which can emphasize the core status of changed code and adaptively learn representations from the context. CodeReviewer pre-trains PLMs based on four proposed pre-training tasks in the code review scenario, which makes CodeReviewer [44] better represent code changes in downstream tasks. Similarly, CCT5 [19] is designed for code-change-related tasks. It proposes five pre-training tasks to build the semantic connection between the changed codes and the corresponding commit messages. In this paper, focusing on the performance that PEFT methods can achieve in code-change-related tasks, we utilize PEFT methods instead of the original FMFT to learn dynamic code semantics.

B. Pre-Trained Language Models of Code

Inspired by the effectiveness and versatility of PLMs shown in the NLP field, a range of PLMs of code that take into account code-specific characteristics has arisen. Although they are all based on Transformer architecture, PLMs of code can be subdivided into three categories: Encoder-only, Decoder-only, and Encoder-Decoder [84], [85]. CodeBERT [1] and GraphCodeBERT [6] are two representative encoder-only models. They are both based on BERT [8] and are pre-trained with natural and programming languages. In addition, GraphCodeBERT introduces semantic-level structural information by adding data flow in the pre-training stage. Decoder-only models of code, like GPT-C [86], CodeGen [87], Code Llama [88], and Qwen2.5-Coder [48], are on the basis of GPT series [89], which predicts text when given the preceding context. Some recent typical encoder-decoder models are PLBART [40], UniXcoder [41], CodeT5 [2], and CodeT5+ [42]. PLBART uses the same architecture as BART [90], and is pre-trained via denoising autoencoding. UniXcoder leverages cross-modal contents like ASTs and code comments to enrich its pre-training corpus, and can be better used for auto-regressive tasks. CodeT5 and CodeT5+ are based on the T5 [91] architecture and propose to preferably capture code semantics via developer-assigned identifiers. In this paper, we apply PEFT methods to a range of PLMs to investigate their performances and generalization abilities.

VIII. Conclusion

For code-change-related tasks, there is an obvious gap between the pre-training and the fine-tuning processes, compared to static code comprehension. Thus, whether PEFT can outperform FMFT is still an open question. In this paper, we experimentally investigate the performance of four prevalent PEFT methods, namely AT, LoRA, PT, and PreT, on two widely-studied code-change-related tasks, including JIT-DP and CMG. Our study shows that PEFT methods can always surpass FMFT on JIT-DP, as well as AT and LoRA can even achieve the SOTA results. But on CMG, although with less training time and memory size consumption, PEFT methods can only exhibit comparable performances at best. While in the cross-lingual and low-resource scenarios, they exhibit relative superiority. To make an explanation for the performance of PEFT methods and FMFT, we also conduct three probing tasks to measure their code semantic learning from both static and dynamic perspectives. Based on the probing results, we propose Pasta$_K$, a new approach that combines the layer-wise advantages of FMFT and PEFT for better domain adaptation. Experimental results show that Pasta$_K$ can significantly improve the performance of PEFT methods while consuming fewer computational resources than FMFT, indicating its practical value. Finally, we summarize our findings and provide implications to enlighten future works.

References

[1] Z. Feng, et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Findings Assoc. Comput. Linguistics (EMNLP)*, 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[2] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, Online and Punta Cana, Dominican Republic, Nov. 2021, pp. 8696–8708. [Online]. Available: https://aclanthology.org/2021.emnlp-main.685

[3] S. Lu et al., "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," 2021, *arXiv:2102.04664*.

[4] P. Xue et al., "Exploring and lifting the robustness of LLM-powered automated program repair with metamorphic testing," 2024, *arXiv:2410.07516*.

[5] A. Karmakar and R. Robbes, "What do pre-trained code models know about code?" in *Proc. 36th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE),* Piscataway, NJ, USA: IEEE Press, 2021, pp. 1332–1336.

[6] D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," in *Proc. 9th Int. Conf. Learn. Representations (ICLR)*, 2021. [Online]. Available: https://openreview.net/forum?id=jLoC4ez43PZ

[7] Y. Xu and Y. Zhu, "A survey on pretrained language models for neural code intelligence," 2022, *arXiv:2212.10079*.

[8] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Human Lang. Technol. (NAACL-HLT)*, 2019, pp. 4171–4186, doi: 10.18653/v1/n19-1423.

[9] N. Houlsby et al., "Parameter-efficient transfer learning for NLP," in *Proc. Int. Conf. Mach. Learn.*, 2019, pp. 2790–2799.

[10] E. J. Hu et al., "LoRA: Low-rank adaptation of large language models," in *Proc. Int. Conf. Learn. Representations*, 2022. [Online]. Available: https://openreview.net/forum?id=nZeVKeeFYf9

[11] X. L. Li and P. Liang, "Prefix-tuning: Optimizing continuous prompts for generation," in *Proc. 59th Annu. Meeting Assoc. Comput. Linguistics 11th Int. Joint Conf. Natural Lang. Process. (Volume 1: Long Papers)*, Online: Association for Computational Linguistics, Aug. 2021, pp. 4582–4597. [Online]. Available: https://aclanthology.org/2021.acl-long.353

[12] B. Lester, R. Al-Rfou, and N. Constant, "The power of scale for parameter-efficient prompt tuning," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, Online and Punta Cana, Dominican Republic: Assoc. for Comput. Linguistics, Nov. 2021, pp. 3045–3059. [Online]. Available: https://aclanthology.org/2021.emnlp-main.243

[13] N. Ding et al., "Parameter-efficient fine-tuning of large-scale pre-trained language models," *Nature Mach. Intell.*, vol. 5, no. 3, pp. 220–235, 2023.

[14] Z. Fu, H. Yang, A. M.-C. So, W. Lam, L. Bing, and N. Collier, "On the effectiveness of parameter-efficient fine-tuning," in *Proc. AAAI Conf. Artif. Intell.*, 2023, vol. 37, pp. 12799–12807.

[15] M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui, "Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 7, pp. 1–25, 2025.

[16] I. Saberi, F. Fard, and F. Chen, "Utilization of pre-trained language model for adapter-based knowledge transfer in software engineering," 2023, *arXiv:2307.08540*.

[17] D. Wang, et al., "One adapter for all programming languages? Adapter tuning for code search and summarization," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, Los Alamitos, CA, USA: IEEE Computer Society, May 2023, pp. 5–16. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE48619.2023.00013

[18] D. Goel, R. Grover, and F. H. Fard, "On the cross-modal transfer from natural language to code through adapter modules," in *Proc. 30th IEEE/ACM Int. Conf. Program Comprehension*, 2022, pp. 71–81.

[19] B. Lin, S. Wang, Z. Liu, Y. Liu, X. Xia, and X. Mao, "CCT5: A code-change-oriented pre-trained model," in *Proc. 31th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng. (ESEC/FSE)*, 2023, pp. 1509–1521.

[20] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? An experimental evaluation of prompt tuning in code intelligence," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 382–394.

[21] R. He, L. Liu, H. Ye, Q. Tan, B. Ding, L. Cheng, J. Low, L. Bing, and L. Si, "On the effectiveness of adapter-based tuning for pretrained language model adaptation," in *Proc. 59th Annu. Meeting Assoc. Comput. Linguistics 11th Int. Joint Conf. Natural Lang. Process. (Volume 1: Long Papers),* Online: Assoc. for Comput. Linguistics, Aug. 2021, pp. 2208–2222. [Online]. Available: https://aclanthology.org/2021.acl-long.172

[22] J. Pfeiffer, A. Rücklé, C. Poth, A. Kamath, I. Vulić, S. Ruder, K. Cho, and I. Gurevych, "AdapterHub: A framework for adapting transformers," in *Proc. Conf. Empirical Methods Natural Lang. Process., Syst. Demonstrations*. Online: Assoc. for Comput. Linguistics, Oct. 2020, pp. 46–54. [Online]. Available: https://aclanthology.org/2020.emnlp-demos.7

[23] T. B. Brown, "Language Models Are Few-Shot Learners," *arXiv preprint, 2020, arXiv:2005.14165*.

[24] Z. Zeng, Y. Zhang, H. Zhang, and L. Zhang, "Deep just-in-time defect prediction: How far are we?" in *Proc. 30th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2021, pp. 427–438.

[25] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *Proc. IEEE 14th Int. Work. Conf. Source Code Anal. Manipulation*, 2014, pp. 275–284.

[26] 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 *Proc. IEEE Int. Conf. Softw. Anal., Evol. Reeng. (SANER)*, 2024, pp. 465–476.

[27] S. Kang, J. Yoon, and S. Yoo, "Large language models are few-shot testers: Exploring LLM-based general bug reproduction," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 2312–2323, doi: 10.1109/ICSE48619.2023.00194.

[28] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "CodaMosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE),* 2023, pp. 919–931.

[29] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 1482–1494, doi: 10.1109/ICSE48619.2023.00129.

[30] Z. Yang et al., "Improving domain-specific neural code generation with few-shot meta-learning," *Inf. Softw. Technol.*, vol. 166, Feb. 2024, Art. no. 107365.

[31] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, vol. 30.

[32] V. Lialin, V. Deshpande, and A. Rumshisky, "Scaling down to scale up: A guide to parameter-efficient fine-tuning," 2023, *arXiv:2303.15647*.

[33] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "Prompt tuning in code intelligence: An experimental evaluation," *IEEE Trans. Softw. Eng.*, vol. 49, no. 11, pp. 4869–4885, Nov. 2023.

[34] C. Feng, Y. Sun, K. Li, P. Zhou, J. Lv, and A. Lu, "Genetic auto-prompt learning for pre-trained code intelligence language models," 2024, *arXiv:2403.13588*.

[35] Y. Gu, X. Han, Z. Liu, and M. Huang, "PPT: Pre-trained prompt tuning for few-shot learning," in *Proc. 60th Annu. Meeting Assoc. Comput. Linguistics (Volume 1: Long Papers).* Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 8410–8423. [Online]. Available: https://aclanthology.org/2022.acl-long.576

[36] J. Liu, C. Sha, and X. Peng, "An empirical study of parameter-efficient fine-tuning methods for pre-trained code models," in *Proc. 38th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE),* Piscataway, NJ, USA: IEEE Press, 2023, pp. 397–408.

[37] Z. Yang et al., "Exploring and unleashing the power of large language models in automated code translation," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 1585–1608, 2024.

[38] S. Feng and C. Chen, "Prompting is all you need: Automated android bug replay with large language models," in *Proc. 46th IEEE/ACM Int. Conf. Softw. Eng.*, 2024, pp. 1–13.

[39] L. Fan, J. Liu, Z. Liu, D. Lo, X. Xia, and S. Li, "Exploring the capabilities of LLMs for code-change-related tasks," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 6, pp. 1–36, 2025.

[40] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Human Lang. Technol.*, Jun. 2021, pp. 2655–2668. [Online]. Available: https://www.aclweb.org/anthology/2021.naacl-main.211

[41] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: Unified cross-modal pre-training for code representation," in *Proc. 60th Annu. Meeting Assoc. Computat. Linguistics (Volume 1: Long Papers)*, Dublin, Ireland, May 2022, pp. 7212–7225. [Online]. Available: https://aclanthology.org/2022.acl-long.499

[42] Y. Wang, H. Le, A. Gotmare, N. Bui, J. Li, and S. Hoi, "Codet5+: Open code large language models for code understanding and generation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2023, pp. 1069–1088.

[43] X. Chen, F. Xu, Y. Huang, N. Zhang, and Z. Zheng, "JIT-smart: A multi-task learning framework for just-in-time defect prediction and localization," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 1–23, 2024.

[44] Z. Li et al., "Automating code review activities by large-scale pre-training," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 1035–1047.

[45] C. Pornprasit and C. K. Tantithamthavorn, "JITLine: A simpler, better, faster, finer-grained just-in-time defect prediction," in *Proc. IEEE/ACM 18th Int. Conf. Mining Softw. Repositories (MSR),* Piscataway, NJ, USA: IEEE Press, 2021, pp. 369–379.

[46] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "Cc2vec: Distributed representations of code changes," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, 2020, pp. 518–529.

[47] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, and D. Lo, "The best of both worlds: Integrating semantic features with expert features for defect prediction and localization," in *Proc. 30th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2022, pp. 672–683.

[48] B. Hui et al., "*Qwen2.5-coder technical report*," 2024, *arXiv:2409.12186*.

[49] M. Zhu, A. Jain, K. Suresh, R. Ravindran, S. Tipirneni, and C. K. Reddy, "XLCoST: A benchmark dataset for cross-lingual code intelligence," 2022, *arXiv:2206.08474*.

[50] R. Chopra, S. Roy, and R. Malhotra, "Transductive instance transfer learning for cross-language defect prediction," in *Proc. 4th Asia Pacific Inf. Technol. Conf.*, 2022, pp. 176–182.

[51] I. Paul, G. Glavaš, and I. Gurevych, "IRCoder: Intermediate representations make language models robust multilingual code generators," 2024, *arXiv:2403.03894*.

[52] D. Pigott, "Online historical encyclopaedia of programming languages," 2020 [Online]. Available: https://hopl.info/home.prx

[53] L. A. Meyerovich and A. S. Rabkin, "Empirical analysis of programming language adoption," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Lang. Appl.,* 2013, pp. 1–18.

[54] N. Zhang et al., "Differentiable prompt makes pre-trained language models better few-shot learners," in *Proc. Int. Conf. Learn. Representations*, 2022. [Online]. Available: https://openreview.net/forum?id=ek9a0qIafW

[55] I. Tenney, D. Das, and E. Pavlick, "BERT rediscovers the classical NLP pipeline," in *Proc. 57th Annu. Meeting Assoc. Computat. Linguistics*, Florence, Italy: Association for Computational Linguistics, Jul. 2019, pp. 4593–4601. [Online]. Available: https://aclanthology.org/P19-1452

[56] I. Tenney et al., "What do you learn from context? probing for sentence structure in contextualized word representations," in *Proc. Int. Conf.*

Learn. Representations, 2019. [Online]. Available: https://openreview.net/forum?id=SJzSgnRcKX

[57] P. Xue et al., "Automated commit message generation with large language models: An empirical study and beyond," *IEEE Trans. Softw. Eng.*, vol. 50, no. 12, pp. 3208–3224, Dec. 2024.

[58] S. Herbold et al., "A fine-grained data set and analysis of tangling in bug fixing commits," *Empirical Softw. Eng.*, vol. 27, no. 6, p. 125, 2022.

[59] Y. Kamei et al., "A large-scale empirical study of just-in-time quality assurance," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, Jun. 2013.

[60] A. Abdu et al., "Semantic and traditional feature fusion for software defect prediction using hybrid deep learning model," *Scientific Rep.*, vol. 14, no. 1, 2024, Art. no. 14771.

[61] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE),* Piscataway, NJ, USA: IEEE Press, 2013, pp. 422–431.

[62] W. Tao, et al., "On the evaluation of commit message generation models: An experimental study," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME),* Piscataway, NJ, USA: IEEE Press, 2021, pp. 126–136.

[63] E. Shi et al., "RACE: Retrieval-augmented commit message generation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, Dec. 2022, pp. 5520–5530.

[64] W. Tao, Y. Zhou, Y. Wang, H. Zhang, H. Wang, and W. Zhang, "Kadel: Knowledge-aware denoising learning for commit message generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 5, pp. 1–32, 2024.

[65] G. Jawahar, B. Sagot, and D. Seddah, "What does BERT learn about the structure of language?" in *Proc. ACL 57th Annu. Meeting Assoc. Comput. Linguistics*, 2019, pp. 3651–3657.

[66] P. Martins, R. Achar, and C. Lopes, "50K-C: A dataset of compilable, and compiled, java projects," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, May 2018, pp. 1–5, doi: 10.1145/3196398.3196450.

[67] J. Dong et al., "FIRA: Fine-grained graph-based code change representation for automated commit message generation," in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 970–981.

[68] A. Conneau, G. Kruszewski, G. Lample, L. Barrault, and M. Baroni, "What you can cram into a single vector: Probing sentence embeddings for linguistic properties," in *Proc. 56th Annu. Meeting Assoc. Computat. Linguistics (Volume 1: Long Papers),* 2018, pp. 2126–2136.

[69] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," 2019, *arXiv:1909.09436*.

[70] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics*, 2002, pp. 311–318.

[71] S. Banerjee and A. Lavie, "METEOR: An automatic metric for MT evaluation with improved correlation with human judgments," in *Proc. ACL Workshop Intrinsic Extrinsic Eval. Measures Mach. Transl. Summarization*, 2005, pp. 65–72.

[72] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries," in *Text Summarization Branches Out*, Barcelona, Spain: Association for Computational Linguistics, 2004, pp. 74–81.

[73] S. Liu, D. Guo, J. Zhang, W. Ma, Y. Li, and Y. Liu, "An empirical study of exploring the capabilities of large language models in code learning," *IEEE Trans. Softw. Eng.*, vol. 51, no. 11, pp. 3088–3102, Nov. 2025.

[74] J. Wang et al., "A survey on cross-lingual summarization," *Trans. Assoc. Comput. Linguistics*, vol. 10, pp. 1304–1323, 2022.

[75] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bull.*, vol. 1, no. 6, pp. 80–83, 1945.

[76] J. Wei et al., "Emergent abilities of large language models," *Trans. Mach. Learn. Res.*, vol. 2022, Aug. 2022.

[77] T. Hoang, J. Lawall, Y. Tian, R. J. Oentaryo, and D. Lo, "PatchNet: Hierarchical deep learning-based stable patch identification for the Linux kernel," *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2471–2486, Nov. 2021.

[78] Z. Yang, J. W. Keung, X. Yu, Y. Xiao, Z. Jin, and J. Zhang, "On the significance of category prediction for code-comment synchronization," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 2, pp. 1–41, 2023.

[79] I. I. Brudaru and A. Zeller, "What is the long-term impact of changes?" in *Proc. Int. Workshop Recommendation Syst. Softw. Eng.*, 2008, pp. 30–32.

[80] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE).* Piscataway, NJ, USA: IEEE Press, 2017, pp. 135–146.

[81] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, "Commit message generation for source code changes," in *Proc. IJCAI*, 2019, pp. 26–33.

[82] T. Hoang, H. K. Dam, Y. Kamei, D. Lo, and N. Ubayashi, "DeepJIT: An end-to-end deep learning framework for just-in-time defect prediction," in *Proc. IEEE/ACM 16th Int. Conf. Mining Softw. Repositories (MSR),* Piscataway, NJ, USA: IEEE Press, 2019, pp. 34–45.

[83] Z. Liu, Z. Tang, X. Xia, and X. Yang, "CCRep: Learning code change representations via pre-trained code model and query back," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 17–29, doi: 10.1109/ICSE48619.2023.00014.

[84] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo, "An empirical comparison of pre-trained models of source code," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE),* Los Alamitos, CA, USA, May 2023, pp. 2136–2148. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE48619.2023.00180

[85] 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 *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA),* New York, NY, USA, 2022, pp. 39–51. [Online]. Available: https://doi.org/10.1145/3533767.3534390

[86] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "IntelliCode compose: Code generation using transformer," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Foundations Softw. Eng. (ESEC/FSE),* 2020, pp. 1433–1443, doi: 10.1145/3368089.3417058.

[87] E. Nijkamp et al., "CodeGen: An open large language model for code with multi-turn program synthesis," in *Proc. ICLR*, 2023.

[88] B. Roziere et al., "Code Llama Open foundation models for code," 2023, *arXiv:2308.12950*.

[89] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei ", and I. Sutskever, "Language models are unsupervised multitask learners," 2019. [Online]. Available: https://cdn.openai.com/better-languagemodels/language_models_are_unsupervised_multitask_learners.pdf

[90] M. Lewis, et al., "BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics*, Jul. 2020, pp. 7871–7880. [Online]. Available: https://aclanthology.org/2020.acl-main.703

[91] C. Raffel et al., "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, 2020. [Online]. Available http://jmlr.org/papers/v21/20-074.html