2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 979-8-3503-3066-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/SANER60148.2024.00055

Delving into Parameter-Efficient Fine-Tuning in Code Change Learning: An Empirical Study

Shuo Liu†, Jacky Keung†, Zhen Yang‡*, Fang Liu§*, Qilin Zhou†, and Yihan Liao†
†Department of Computer Science, City University of Hong Kong, Hong Kong, China,
{sliu273-c, qilin.zhou,yihanliao4-c}@my.cityu.edu.hk, Jacky.Keung@cityu.edu.hk
‡ School of Computer Science and Technology, Shandong University, Qingdao, China, zhenyang@sdu.edu.cn
§School of Computer Science and Engineering, Beihang University, Beijing, China, fangliu@buaa.edu.cn

***Abstract*—Compared to Full-Model Fine-Tuning (FMFT), Parameter Efficient Fine-Tuning (PEFT) has demonstrated superior performance and lower computational overhead in several code understanding tasks, such as code summarization and code search. This advantage can be attributed to PEFT's ability to alleviate the catastrophic forgetting issue of Pre-trained Language Models (PLMs) by updating only a small number of parameters. As a result, PEFT effectively harnesses the pre-trained general-purpose knowledge for downstream tasks. 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 two prevalent PEFT methods, namely Adapter Tuning (AT) and Low-Rank Adaptation (LoRA), and compare their performance with FMFT on five popular PLMs. Specifically, we evaluate their performance on two widely-studied code-change-related tasks: Just-In-Time Defect Prediction (JIT-DP) and Commit Message Generation (CMG). The results demonstrate that both AT and LoRA achieve state-of-the-art (SOTA) results in JIT-DP and exhibit comparable performances in CMG when compared to FMFT and other SOTA approaches. Furthermore, AT and LoRA exhibit superiority in cross-lingual and low-resource scenarios. We also conduct three probing tasks to explain the efficacy of PEFT techniques on JIT-DP and CMG tasks from both static and dynamic perspectives. The study indicates that PEFT, particularly through the use of AT and LoRA, offers promising advantages in code-change-related tasks, surpassing FMFT in certain aspects. This research contributes to a deeper understanding of the capabilities of PEFT in leveraging pre-trained PLMs for dynamic code changes. The replication package is available at https://github.com/ishuoliu/PEFT4CC.**

***Index Terms*—Pre-trained Language Models, Adapter Tuning, Low-Rank Adaptation, Code Change**

I. Introduction

Pre-trained Language Models (PLMs), e.g., CodeBERT [9] and CodeT5 [59], have been extensively utilized in various software engineering tasks, including code generation [13], [37] and code comprehension [14], [26], and have achieved notable advancements [62]. These models follow a prevalent paradigm where they are initially pre-trained on large-scale monolingual corpora using Masked Language Modeling (MLM) or Next Token Prediction (NTP) [5], and subsequently fine-tuned for specific downstream tasks. Nonetheless, Full-Model Fine-Tuning (FMFT) prohibitively relies on enormous computational resources, rendering it unsuitable for all users. Therefore, a series of Parameter-Efficient Fine-Tuning (PEFT) methods [20], [21], [28], [30], updating only 1%-5% of the whole parameters while keeping the others frozen, have been proposed in recent years and gained much attention [6], [10].

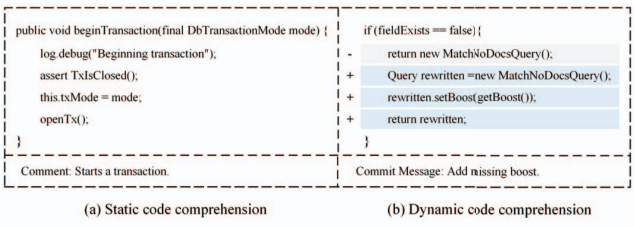


Fig. 1. Examples of static and dynamic code comprehension.

Previous studies [48], [57] 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 to the downstream tasks [11]. However, it is noted that most of the investigation objects of the above literature are static code comprehension tasks, excluding dynamic code changes, which is consistent with the paradigm that PLMs learned during their pre-training phase. As such, we conjecture adapting PLMs to static code comprehension tasks inherently needs fewer parameter tuning. However, for downstream tasks concerning code changes, both before and after-change code snippets are involved, enlarging the transferring gap between the pre-training and adaptation phases [33], [56]. Figure 1 shows examples of static and dynamic code comprehension, where the former mainly relates to a single status of a code comment pair, while the latter contains a code *diff* that compares program statuses of old and new versions line by line, described by a commit message recording the intent of changes. Considering their discrepancies above, whether PEFT still outperforms FMFT in code-change-related tasks remains an open question.

In this paper, we explore the effect of PEFT on code change learning. We choose two mainstream PEFT methods for experiments, namely Adapter Tuning (AT) [15], [20], [43] and Low-Rank Adaptation (LoRA) [21]. AT introduces adapter modules, each of which is a bottleneck structure, and inserts them between layers of PLMs. In the fine-tuning

*Corresponding author.

2640-7574/24/$31.00 ©2024 IEEE
DOI 10.1109/SANER60148.2024.00055

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. To evaluate their performance on code change learning, we conduct experiments for Just-In-Time Defect Prediction (JIT-DP) [66] and Commit Message Generation (CMG) [4]. 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, we first compare the performance of AT and LoRA with FMFT and state-of-the-art approaches. Furthermore, to evaluate how well AT and LoRA transfer knowledge and generalize across languages, we explore their capabilities in the cross-lingual scenario, where PLMs are fine-tuned in one programming language but are tested in another. Besides, it is widely acknowledged that the performance of FMFT is highly dependent on the amount of available downstream data. In order to understand how well PEFT performs in scenarios where data availability is limited, we conducted experiments to examine its effectiveness in handling situations of data scarcity, where the fine-tuned datasets are scaled down, but the testing datasets remain the same. We also utilize three probing tasks, namely Invalid Type Detection (TYP), Code Change Match (CCM), and Line Type Prediction (LTP), to investigate the encoded code semantics, expecting to make an explanation for the performance of PEFT. TYP is related to the static code semantics, while CCM and LTP are associated with the dynamic code semantics from global and local perspectives, respectively. Our experimental results show that, in the JIT-DP task, AT and LoRA obtain improvements of 8.39% and 9.87% in terms of F1 when compared with the State-Of-The-Art (SOTA) baseline, and in the CMG task, they perform a little weakly. However, considering the less training time and memory consumption, PEFT techniques are still practical and acceptable. Besides, even in the cross-lingual and low-resource scenarios, they also exhibit effectiveness and superiority. Three probing tasks concerning both static and dynamic code semantics explain the efficacy of PEFT techniques and FMFT.

The major contributions of this paper are as follows:

1. To the best of our knowledge, this paper serves as the first study to explore the performance of PEFT on code-change-related tasks.
2. We conduct extensive experiments, involving two PEFT methods, five 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.

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 [25], [27], [60], [63]. 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:

$$\Theta = \Theta_0 + \Delta\Theta = \arg\min_{\Theta} L(X;\Theta) \tag{1}$$

where L is a loss function and $\Delta\Theta$ represents the adjustment of parameters. Considering that PLMs are Transformer-based architecture [55], FMFT is illustrated in Figure 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.

B. Adapter Tuning

Adapter tuning is a representative method of PEFT, which adds extra compact modules (adapters) to every transformer layer [15], [20], [43]. Figure 2(b) shows the standard architecture of adapter tuning [20], 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 adapter tuning is to train the added modules and corresponding following layer normalizations. Its tuning objective can be defined as:

$$\Phi_a = \arg\min_{\Phi_a} L(X;\Theta_0,\Phi_a) \tag{2}$$

where Φ_a is the adapter parameter. X and L are task-specific dataset and loss function respectively. PLMs' pre-trained weights Θ_0 are frozen here. During adapter tuning, around 3% - 5% of the whole parameters are trainable, which is considerably fewer than fine-tuning.

C. Low-Rank Adaptation (LoRA)

Different from adapter tuning that adds extra trainable parameters, LoRA utilizes the idea of reparameterization to learn the parameters' updation by low-rank matrices [32]. As illustrated in Figure 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 [21]. 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:

$$\Phi_l = \arg\min_{\Phi_l} L(X;\Theta_0 + \Delta\Theta(\Phi_l)) \tag{3}$$

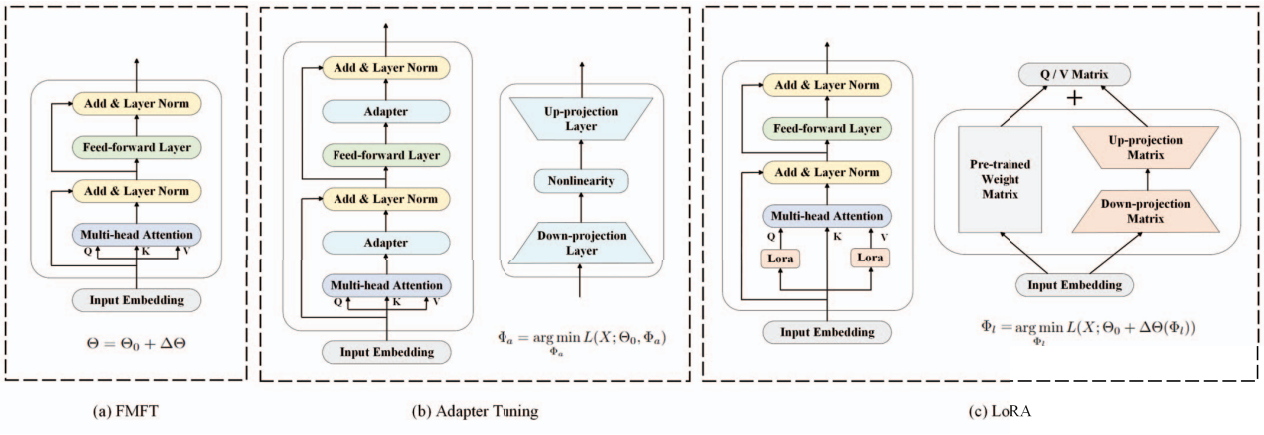


Fig. 2. Illustrations on fine-tuning, adapter tuning, and low-rank adaptation.

where X and L are the dataset and loss function of downstream tasks. 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%.

III. Experimental Setup

A. Research Questions

Although previous studies have shown the effectiveness of PEFT in code search, code clone, and code summarization [48], [57], these tasks are limited in the comprehension of static code semantics. For code-change-related tasks that require the understanding of dynamic code semantics, there is still an open question about the efficacy of PEFT. To address the above issue, we investigate the following research questions about adapter tuning and LoRA, exploring their effects and advantages.

RQ1: How do adapter tuning and LoRA perform compared with FMFT and SOTA approaches? Previous studies have shown the effectiveness of PEFT methods in static code comprehension tasks, we investigate whether they can perform consistent advantages in code-change-related tasks. Specifically, we compare the performance of FMFT, AT, and LoRA on two widely studied code-change-related tasks, 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 AT and LoRA to five popular PLMs, namely CodeBERT, GraphCodeBERT, PLBART, UniXcoder, and CodeT5, to conduct comprehensive comparisons and explore the generalization abilities of PEFT on diverse PLMs. Besides, various task-specific SOTA approaches are also involved for comparison [18], [33], [39], [44]. For CMG, We apply the three methods on CodeT5 because it is the backbone model of current SOTA approaches [31], [33].

RQ2: How capable are adapter tuning and LoRA in the cross-lingual scenario compared with FMFT? In the cross-lingual scenario, pre-trained PLMs are fine-tuned in one Programming Language (PL) and are evaluated in another. It is a common but resultful approach to relieve the problem of data scarcity [68]. To explore the performance of adapter tuning and LoRA in the cross-lingual scenario, we carry out experiments with CodeT5 on the CMG task, which involves five PLs. For each PL, we attempt FMFT, AT, and LoRA for adaptation, then evaluate them on each of the other PLs.

RQ3: How capable are adapter tuning and LoRA in the low-resource scenario compared with FMFT? The performance of FMFT prohibitively relies on the scale of downstream data [12], [28], [67]. 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 that the CMG task contains hundreds of thousands of data, we investigate the performance of adapter tuning and LoRA when randomly sampling 1,000, 5,000, and 10,000 training data for each PL. We still evaluate the models on the original validation set and testing set.

RQ4: What kind of knowledge is encoded by adapter tuning and LoRA? To explore what benefits adapter tuning and LoRA can bring about, we utilize probing tasks, which have been extensively used in the NLP field [53], [54], to understand the encoded linguistic information. We employ three probing tasks, invalid type detection for static semantic-level information [26], as well as code change match and line type prediction, 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 adapter tuning and LoRA, as well as making comparisons with FMFT, thereby providing explanations for their performance.

B. Tasks and Datasets

1) Just-In-Time Defect Prediction: The change of code may damage software quality, so it is crucial to discover defects as early as possible [66]. 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 for the outputs are two classes, namely defective or not.

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

The experimental dataset we used in JIT-DP is JIT-Defects4J [39], which is extended from LLTC4J [16] 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. [24], which can measure the code change from five dimensions, i.e. diffusion, size, purpose, history, and experience. Following previous studies [33], [39], 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.

2) 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 [4]. However, writing high-quality commit messages is time-consuming, and thus many are left empty [8]. 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.

In the CMG task, we choose the Multi-language Commit Message Dataset (MCMD) for experiments [52]. 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. Furthermore, following the operation of Shi et al. [49], noisy data containing multiple files or files that cannot be parsed are filtered out. The statistics of MCMD are shown in Table I.

3) Probing Tasks: 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. [26], aims to measure the static code semantics understood by PLMs. As Figure 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. [26]. It is gathered from 50K-C [38], 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 a commit message corresponds to a given code change. It is shown in Figure 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 [7], 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 the illustration in Figure 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 [7]. 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.

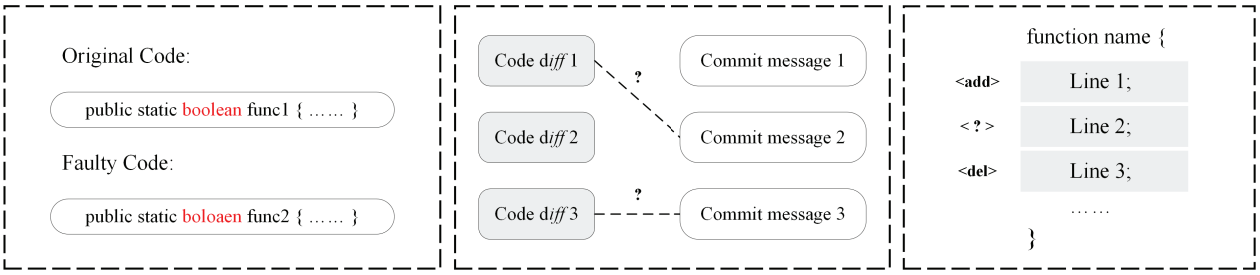
C. Pre-trained Language Models and Baselines

There are five PLMs that we use to evaluate the performance of FMFT, adapter tuning, and LoRA. In the JIT-DP task, we conduct experiments on all of the five PLMs, where only encoders are used for the purpose of code change representation. Besides, we choose CodeT5 as the backbone model for the CMG task because it is used by the SOTA baselines [31], [33]. The details of the PLMs are described in the following:

- **CodeBERT** [9]: It is an encoder-only PLM with 125M parameters, pre-trained on CodeSearchNet [22] which contains 2M bimodal data.
- **GraphCodeBERT** [14]: 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** [1]: 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** [13]: It is a unified encoder-decoder PLM with 125M parameters, and can be flexibly modified to encoder-only or decoder-only architecture.
- **CodeT5** [59]: It contains 220M parameters, and is a representative PLM of encoder-decoder architecture for its well performance in generation tasks.

We also compare the performance of FMFT, adapter tuning, and LoRA with the following baselines.

- **JITLine** [44]: 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** [39]: 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.



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

Fig. 3. Illustrations of probing tasks.

- **CC2Vec** [18]: 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** [31]: 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 [59] capacity for understanding code changes.
- **CCT5** [33]: It is the state-of-the-art baseline for both JIT-DP and CMG. It also pre-trains CodeT5 [59] with code-change-related corpus and objectives.

D. Evaluation Metrics

1) Just-In-Time Defect Prediction: Following prior works [33], [39], we use the F1-score and the Area Under the receiver operating characteristic Curve (AUC) as the evaluation metrics. 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 [42], Meteor [2], and Rouge-L [35]. 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, we use accuracy as the evaluation metric, which measures the correctness of predictions.

E. Implementation Details

The experiments were conducted on a Ubuntu GPU server with two RTX 3090 24GB GPUs. Our code is implemented by the deep learning framework PyTorch[1]. The PLMs we used are from Huggingface[2], and we keep their default hyperparameter settings. For the adapter tuning, we implement it with the OpenDelta Library[3], and we set the intermediate dimension m to 128. LoRA is implemented by the PEFT Library[4], and its dimension r is 8. In the JIT-DP task, we adjust the learning rate in the range of {1e-3, 5e-4, 1e-4, 5e-5} for all PLMs. In the CMG task, we set the learning rate of CodeT5 to 5e-4 and keep it consistent across FMFT, adapter tuning, and LoRA. Besides, we set the maximum training epoch to 10, and adopt early stopping with the patience of 5. Due to the performance of baselines on our evaluation datasets having been investigated in previous studies, we reuse the performance reported by Ni et al. [39] and Lin et al. [33].

[1]https://pytorch.org/
[2]https://huggingface.co/models
[3]https://opendelta.readthedocs.io/en/latest/

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
	Avg.	0.380	0.852	0.421	0.873
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
	Avg.	0.383	0.862	**0.519**	**0.898**
Adapter Tuning	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
	Avg.	0.394	0.865	0.512	0.895

IV. EXPERIMENTAL RESULTS

A. RQ1: Quantitative Evaluation

1) Just-In-Time Defect Prediction: We first evaluate the performance of adapter tuning and LoRA in the JIT-DP task. We employ five prevalent PLMs to exhibit comprehensive comparisons, including CodeBERT, GraphCodeBERT, PLBART, UniXcoder, and CodeT5. The latter three PLMs are encoder-decoder architecture, and we only use their encoders in the task. Following previous studies [33], [39], we also

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

conduct experiments on two settings. The first is a common setting (w/o EF), where 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 extra expert features [24], and the concatenated feature vectors are used to predict. Table II shows the detailed evaluation results. Because they do not utilize expert features in their original papers, the performance of JITLine and CC2Vec in the second setting is omitted.

When comparing the performance of FMFT, adapter tuning, and LoRA, it can be noticed that no matter in which setting, the average performance of adapter tuning and LoRA is better than FMFT. Specifically, in the common setting, adapter tuning obtains an improvement of 3.58% in terms of F1 and 1.57% in terms of AUC, while LoRA increases by 0.84% in terms of F1 and 1.22% in terms of AUC. When incorporating extra expert features, the improvements become significant. Adapter tuning surpasses fine-tuning by 21.52% and 2.52% for F1 and AUC, respectively. LoRA increases 23.18% and 2.86% accordingly. The results indicate that adapter tuning and LoRA show their effectiveness in the JIT-DP task, especially when introducing extra expert features. A potential explanation is adapter tuning and LoRA 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 adapter tuning and LoRA consistently show their improvements in diverse PLMs, which manifests their generalization abilities.

When compared with other baselines, adapter tuning and LoRA achieve state-of-the-art results that reach 0.512 and 0.519 in terms of F1 in the setting with extra expert features. They surpass CCT5 [33], the current SOTA method, by 8.39% and 9.87%, respectively. Considering that CCT5 has to be pre-trained from scratch on code-change-related corpus and various objectives, it indicates the prominent advantage of adapter tuning and LoRA because they only need to fine-tune a very small portion of the parameters.

> **Finding 1**: Compared with FMFT, Adapter tuning and LoRA show their effectiveness in the JIT-DP task, especially when incorporating extra expert features. They obtain state-of-the-art results that achieve improvements of 8.39% and 9.87% in terms of F1 sore compared with the SOTA approach, respectively.

2) Commit Message Generation: We also evaluate adapter tuning and LoRA on a widely-studied code-change-related generation task, namely commit message generation. We employ CodeT5 as the backbone PLM following previous studies [31], [33]. 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. Because CodeReviewer and CCT5 utilize BLEU to evaluate their models, their performance on the Meteor and Rouge-L is omitted here.

From Table III, we could observe mild degradations of

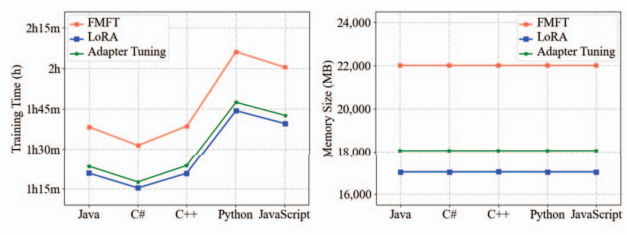


Fig. 4. Comparison of training time and memory size.

adapter tuning and LoRA when comparing them with FMFT. The degradation is consistent on the five PLs. The average performance of adapter tuning and LoRA are 21.60 and 19.85 in terms of BLEU, both of which are lower than the FMFT performance of 22.16. We conjecture the reasons for such decrements mainly contain: (i) In the architecture of CodeT5, adapter tuning and LoRA only modify 4.10% and 0.40% of the full parameters, which is quite a small portion of parameters. (ii) Compared with classification tasks, e.g., the JIT-DP task, generative tasks, such as the CMG task, carry a high complexity and difficulty. As such, the CMG task needs more parameter tuning for knowledge adaptation. The performance of adapter tuning is better than LoRA in the CMG task, and we attribute it to the more adapted parameters that adapter tuning has. In addition to evaluating the performance, we analyze the training time and memory size of the three methods, and the results are illustrated in Figure 4. We report the average training time of each epoch and the occupied memory size on a GPU. The devices we use are elaborated in Section III-E. It can be observed that adapter tuning and LoRA can save approximately 15-20 minutes each epoch, and decrease memory consumption of about 4,000-5,000 MB. It indicates that they can achieve similar performance compared with FMFT, but in less training time and memory consumption.

Although CodeReviewer and CCT5 are pre-trained on code change datasets, when compared with them, adapter tuning and LoRA still perform well. They both surpass CodeReviewer on the average performance. Besides, adapter tuning outperforms CCT5 on two sub-datasets, Java and C++. It is well known that pre-training on large-scale datasets is not practical for everyone, but adapter tuning and LoRA can obtain similar results by modifying only a small portion of the parameters.

> **Finding 2**: In the CMG task, adapter tuning and LoRA can achieve similar performances compared with fine-tuning and state-of-the-art baselines, while they consume less training time and memory size.

B. RQ2: Performance in the Cross-lingual Scenario

In this section, we investigate the performance of adapter tuning and LoRA in the cross-lingual scenario, comparing them with FMFT as well. We apply the three 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 [58], and is useful when the

TABLE III
EVALUATION RESULTS OF CMG ON THE MCMD DATASET. M IS SHORT FOR METEOR. R IS SHORT FOR ROUGE-L. CODET5, CODET5-L, AND CODET5-A REPRESENT THEY UTILIZE FMFT, ADAPTER TUNING, AND LORA, RESPECTIVELY.

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	18.47	-	-	20.36	-	-	15.94	-	-	17.65	-	-	19.84	-	-	18.45	-	-
CCT5	20.80	-	-	**25.53**	-	-	17.64	-	-	**21.37**	-	-	24.94	-	-	22.06	-	-
CodeT5	**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**
CodeT5-L	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
CodeT5-A	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

TABLE IV
EVALUATION RESULTS OF CMG IN THE CROSS-LINGUAL SCENARIO.

Models		Java	C#	C++	Python	JavaScript	Avg.
Java	CodeT5	-	9.94	10.06	10.95	12.10	10.76
	CodeT5-L	-	**10.52**	**11.00**	**11.70**	**13.30**	**11.63**
	CodeT5-A	-	10.47	10.69	11.31	12.90	11.34
C#	CodeT5	10.23	-	9.97	10.68	13.18	11.02
	CodeT5-L	**11.01**	-	**11.04**	**11.57**	13.21	**11.71**
	CodeT5-A	10.65	-	10.89	11.44	**13.34**	11.58
C++	CodeT5	10.64	10.42	-	12.08	13.44	11.65
	CodeT5-L	**11.26**	**10.93**	-	**12.64**	**14.20**	**12.26**
	CodeT5-A	11.00	10.31	-	12.34	13.52	11.79
Python	CodeT5	9.78	10.14	10.47	-	13.24	10.91
	CodeT5-L	**10.20**	10.54	10.99	-	**13.79**	**11.38**
	CodeT5-A	10.00	**10.55**	**11.03**	-	13.69	11.32
JavaScript	CodeT5	9.62	9.90	9.75	10.93	-	10.05
	CodeT5-L	**11.37**	**10.73**	**11.20**	**12.30**	-	**11.40**
	CodeT5-A	10.65	9.96	10.63	11.88	-	10.78

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, which represent source PL and target PL are the same, we could observe consistent improvements brought by adapter tuning and LoRA in the cross-lingual scenario. When comparing with the performance of FMFT, the average promotion of adapter tuning and LoRA can be up to 4.62% and 7.56%, respectively. It indicates that FMFT learns much PL-specific knowledge to pursue high performance, while adapter tuning and LoRA adjusting a small portion of parameters learn general-purpose knowledge, which can be transferred in diverse PLs. In addition, LoRA outperforms adapter tuning in most cross-lingual scenarios. Considering their parameter size, the result inspires us that tuning very few parameters can keep a balance between good performance and generality among different PLs, which is the superiority of adapter tuning and LoRA.

Finding 3: Adapter tuning and LoRA obtain consistent improvements than FMFT in the cross-lingual scenario. It shows their superiority in balancing PLMs' high performance and generality among different PLs.

C. RQ3: Performance in the Low-resource Scenario

Low resource is also a common scenario for code intelligence tasks [50]. In this section, we investigate how well adapter tuning and LoRA can perform in the low-resource scenario. We randomly select 1,000, 5,000, and 10,000 training data for each PL in the CMG task, and the evaluation results are shown in Table V. From Table V, when training with 10,000 samples ($<$ 1/10 of the full-data setting) for each PL, except for C++, it can be observed that adapter tuning and LoRA consistently outperform FMFT in the other four PLs. As for their average performance, adapter tuning obtains promotions of 5.45%, 4.60%, and 5.86% for BLEU, Meteor, and Rouge-L, respectively. Accordingly, LoRA surpasses FMFT by 0.20%, 3.17%, and 1.48% in each of the metrics, respectively. This demonstrates that PEFT techniques make PLMs adapt to code-change-related tasks more efficiently compared with FMFT. Nonetheless, when reducing the training samples to 1,000 and 5,000 for each PL ($<$ 1/100 and $<$ 1/50 of the full-data setting), it can be observed that the average performance of FMFT becomes slightly better than adapter tuning and LoRA. It is also noted that, in the setting of 1,000 training samples, adapter tuning and LoRA outperform FMFT in C++, and in the setting of 5,000 training samples, adapter tuning and LoRA surpass FMFT in Python and JavaScript. Compared with the full-data setting where FMFT always outperforms PEFT techniques slightly (refer to RQ1), PEFT techniques carry their superiority in the low-resource setting. We speculate that CMG is a tough code-change-related task that needs more parameter tuning for adaptation. Even though PEFT inherently carries superiority in the low-resource setting, they still cannot exhibit a better understanding of code changes with very limited training samples in such generative tasks. It reminds us that PEFT needs a certain amount of data to learn such tricky code-change-related semantics. However, as the performance differences between PEFT techniques and FMFT are still comparable, we still suggest using PEFT techniques in the low-resource setting, as their low computational overhead.

Finding 4: Adapter tuning and LoRA show effectiveness in the low-resource scenario. Practitioners are recommended to use PEFT techniques in the low-resource setting as their performance is outperforming or at least comparable to FMFT.

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. To be specific, in the JIT-DP task, we reuse the five PLMs that are trained with expert features by FMFT, adapter

TABLE V
EVALUATION RESULTS OF CMG IN THE LOW-RESOURCE SCENARIO. M IS SHORT FOR METEOR. R IS SHORT FOR ROUGE-L. CODET5, CODET5-L, CODET5-A REPRESENT THEY UTILIZE FMFT, ADAPTER TUNING, AND LORA, RESPECTIVELY.

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	CodeT5	9.68	**7.21**	**13.91**	**9.34**	**6.04**	**12.72**	7.59	5.73	10.54	**9.72**	6.92	12.66	**13.25**	**8.66**	**17.88**	**9.92**	**6.91**	**13.54**
	CodeT5-L	8.89	6.30	12.50	7.78	4.91	10.45	**8.85**	**6.43**	**11.24**	9.40	**7.15**	**12.99**	11.12	8.22	16.35	9.21	6.60	12.71
	CodeT5-A	**9.71**	6.33	13.21	7.99	4.64	10.93	8.18	5.92	11.23	8.45	6.33	11.63	10.76	7.72	15.36	9.02	6.19	12.47
5,000	CodeT5	**14.95**	9.48	**20.09**	**13.53**	**9.15**	**19.14**	11.71	**9.52**	**17.21**	12.37	9.03	17.29	14.96	10.43	20.15	**13.50**	**9.52**	**18.78**
	CodeT5-L	13.96	9.46	18.96	12.22	7.56	16.55	11.50	8.66	15.74	**12.71**	**9.85**	**17.95**	14.94	**11.03**	**21.11**	13.07	9.31	18.06
	CodeT5-A	14.55	**10.18**	19.91	12.55	7.85	17.14	**11.84**	8.71	15.68	12.59	9.37	17.28	**15.05**	10.71	20.49	13.32	9.36	18.10
10,000	CodeT5	16.07	10.35	21.47	13.00	8.11	17.36	**13.38**	**9.86**	**18.19**	13.49	10.19	18.73	17.88	12.32	24.18	14.76	10.17	19.99
	CodeT5-L	16.13	10.61	21.62	13.76	9.05	18.73	12.77	9.45	17.15	13.81	10.70	19.85	17.50	12.63	24.06	14.79	10.49	20.28
	CodeT5-A	**17.46**	**11.38**	**23.41**	**14.37**	**9.64**	**19.61**	12.93	9.82	18.02	**14.73**	**11.13**	**21.28**	**18.51**	**12.88**	**25.03**	**15.60**	**10.97**	**21.47**

TABLE VI
EVALUATION RESULTS OF PROBING TASKS FOR JIT-DP. THE BEST AVERAGE PERFORMANCE IS HIGHLIGHTED IN BOLDFACE.

	Models	TYP	CCM	LTP
FMFT	CodeBERT	54.0	50.0	62.5
	GraphCodeBERT	78.5	58.0	71.5
	PLBART	80.0	62.0	73.5
	UniXcoder	89.0	52.5	65.5
	CodeT5	91.5	70.0	79.5
	Avg.	78.6	58.5	70.5
LoRA	CodeBERT	91.0	50.5	72.5
	GraphCodeBERT	97.5	61.5	71.5
	PLBART	87.5	55.5	73.5
	UniXcoder	95.5	58.5	67.5
	CodeT5	92.5	64.5	77.0
	Avg.	**92.8**	58.1	72.4
Adapter Tuning	CodeBERT	91.0	56.0	65.0
	GraphCodeBERT	94.0	61.0	71.0
	PLBART	84.5	63.0	76.5
	UniXcoder	93.5	55.5	70.5
	CodeT5	95.5	72.0	80.5
	Avg.	91.7	**61.5**	**72.7**

tuning, and LoRA, respectively. We also probe the fine-tuned PLM (i.e., CodeT5) in the CMG task among different PLs. All PLMs used here are trained in the full-data setting. Taking probing tasks' data as inputs, we extract the encoded features from the last hidden layer, which is the 6-*th* layer for PLBART and the 12-*th* layer for the others. Then extracted features are fed into a simple linear classifier to predict their categories [26]. The classifier has no hidden units, so the performance of probing tasks prohibitively depends on the feature vectors. The evaluation results of probing tasks for JIT-DP and CMG are illustrated in Table VI and Table VII.

From Table VI, we could derive several insightful findings: (i) The average performance of adapter tuning outperforms FMFT in all three probing tasks, while the average performance of LoRA surpasses FMFT in TYP and LTP, but degrades slightly in CCM. (ii) Comparing with the improvements obtained by adapter tuning and LoRA in the TYP task, the promotions on CCM and LTP are not dramatic. The first finding is expected and explains that the adapter tuning and LoRA 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. In addition, we also extract encoded features from each layer of CodeBERT to probe the layer-wise performance, as shown in the first row of Figure 5. The dashed lines indicate the average performance of all layers. Except for the adapter tuning in the LTP task, consistent substantial improvements are obtained by adapter tuning and LoRA, which also shows their superiority in representing both static and dynamic code semantics.

As for the CMG task, Table VII shows that adapter tuning and LoRA can not surpass FMFT in all three probing tasks on average of diverse PLs, but their differences are not evident. Subsequently, we further select Java to dig deeper into the layer-wise probing, as shown in the second row of Figure 5. We found that no one approach can dominate all three probing tasks. Instead, they achieve an almost neck-to-neck performance on average of different layers. For example, FMFT and PEFT techniques perform almost the same in the TYP task. Adapter tuning outperforms the other two approaches in the CCM task while FMFT outperforms them in the LTP task. Hence, it is hard to say which approach makes PLMs learn more about static or dynamic code semantics. To some extent, it is also consistent with the experimental results of RQ1, where the CMG ability learned by three techniques is comparable.

Finding 5: Probing tasks show that adapter tuning and LoRA can benefit the understanding of dynamic code semantics from both the global and local perspectives in the JIT-DP task. In the CMG task, probing tasks show the differences among FMFT, adapter tuning, and LoRA are not evident.

V. IMPLICATIONS

This paper presents the first empirical study that investigates the performance of PEFT on code-change-related tasks. In this section, we discuss some 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, and can achieve comparable results on the CMG task with less training time and memory consumption. The results enlighten developers 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

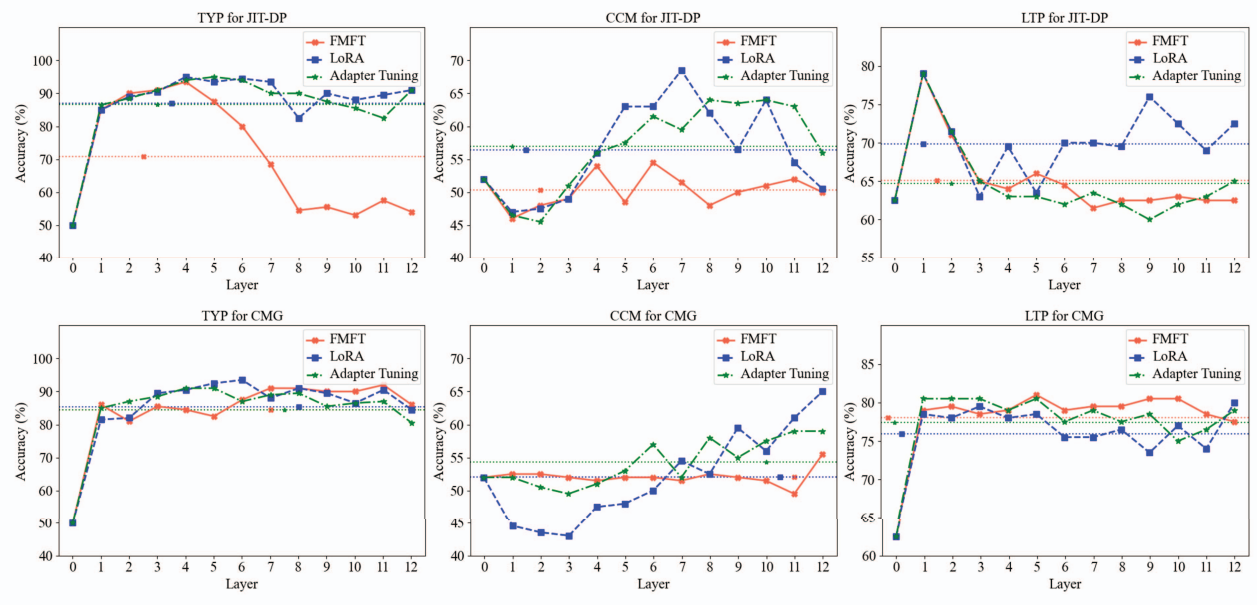


Fig. 5. Accuracy of probing tasks by layers.

TABLE VII
EVALUATION RESULTS OF PROBING TASKS FOR CMG. THE BEST AVERAGE PERFORMANCE IS HIGHLIGHTED IN BOLDFACE.

Models		TYP	CCM	LTP
FMFT	Java	86.0	55.5	77.5
	C#	88.0	57.0	78.0
	C++	85.5	57.5	79.0
	Python	86.0	51.0	75.5
	JavaScript	88.5	54.0	79.0
	Avg.	86.8	55.0	77.8
LoRA	Java	84.5	65.0	80.0
	C#	91.0	54.5	77.5
	C++	90.0	58.0	77.5
	Python	88.5	58.5	76.5
	JavaScript	86.5	60.5	73.5
	Avg.	**88.1**	59.3	77.0
Adapter Tuning	Java	80.5	59.0	79.0
	C#	83.5	62.0	83.5
	C++	84.0	68.5	81.0
	Python	81.5	65.0	83.5
	JavaScript	92.0	63.0	84.0
	Avg.	84.3	**63.5**	**82.2**

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 [48], [57], our work provides guidance about when and how to apply PEFT to code-change-related tasks.

Implications for researchers. As the first empirical study to explore the performance of PEFT in code-change-related tasks, this work obtains several interesting findings. It can be seen that in the JIT-DP task, the promotion of PEFT increases when adding the expert features, which demonstrates that PEFT can facilitate encoding the extra features into PLMs. Because expert features are more straightforward to code-change-related semantics, adjusting few parameters can transfer them to specific downstream tasks. It deserves to be investigated in other tasks to pursue high performances. We also call for more effective probing tasks designed for code-change-related tasks, probing information like syntax and structure. Besides, our study demonstrates that PEFT can be a powerful alternative approach, with their standard architectures. It also represents the need to further adapt PEFT with more code-change-related designs.

VI. THREATS TO VALIDITY

We identify the following threats to our study: **1) Evaluation Tasks**: We conduct experiments on two widely-studied code-change-related tasks, including a classification task, JIT-DP, and a generation task, CMG. Although they are representative, there are other code-change-related tasks like bug fixing patch identification [19] and automated patch correctness assessment [34]. Therefore, in the future, we plan to assess our findings on more tasks. **2) Evaluation Datasets**: The quality of datasets can affect the evaluation performance. Thus, to mitigate the issue, we select widely-used large-scale datasets for JIT-DP and CMG. However, it is noted that there are also other datasets constructed from different source data. We will conduct experiments on more datasets to confirm our findings in future works. **3) Pre-trained Language Models**: Due to the computational resource constraints, the maximal PLM we use in our study is CodeT5, which contains 220M parameters. In the future, larger PLMs deserve to be explored. **4) Hyperparameter Setting**: In this study, we implement the standard architecture of AT and LoRA and follow their default hyperparameter settings. We also modify their intermediate dimensions, which does not bring expected improvement. Finding optimal hyperparameters is always a difficult task that is needed to explore 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 [3], [64]. Along with code changes, developers have to undertake considerable workloads for tasks like writing high-quality commit messages [23] and identifying whether software changes are defect-inducing [24]. 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 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 [61] extracts code structure and code semantics by separate bidirectional GRU and aligns the two parts by an attention layer. FIRA [7] describes code change operations in fine-grained graphs and utilizes a graph-neural-network-based encoder to learn representations. RACE [49], based on 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 [17] leverages two discrete CNNs to extract features from code changes and corresponding commit messages, and employs a fully connected network for feature fusion. JITLine [44] 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 [39] combines 14 change-level expert features [24] with extracted semantic features, obtaining integrated representations.

There are also some works focusing on general-purpose code change representations. CC2Vec [18] 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 [36] 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 [31] better represent code changes in downstream tasks. Similarly, CCT5 [33] is designed for code-change-related tasks. It proposes five pre-training tasks to build the semantic connection between the changed codes and 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 arise. Although they all based on Transformer architecture, PLMs of code can be subdivided into three categories: Encoder-only, Decoder-only, and Encoder-Decoder [41], [65]. CodeBERT [9] and GraphCodeBERT [14] are two representative encoder-only models. They are both based on BERT [5] 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 [51], CodeGen [40], and Code Llama [47], are on the basis of GPT series [45], which predicts text when given the preceding context. Some recent typical encoder-decoder models are PLBART [1], UniXcoder [13], and CodeT5 [59]. PLBART uses the same architecture as BART [29], 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 is based on T5 [46] architecture and proposes 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 two prevalent PEFT methods, namely AT and LoRA, on two widely-studied code-change-related tasks, including JIT-DP and CMG. Our study shows that AT and LoRA can achieve the SOTA results on JIT-DP and comparable performances on CMG with less training time and memory consumption, which indicates their effectiveness and efficiency. Even in the cross-lingual and low-resource scenarios, they also exhibit superiority. To make an explanation for the performance of AT, LoRA, and FMFT, we also conduct three probing tasks to measure their code semantic learning from both static and dynamic perspectives. Finally, we summarize our findings and provide implications to enlighten future works.

Acknowledgment

This work is supported in part by the General Research Fund (GRF) of the Research Grants Council of Hong Kong, and the industry research funds of City University of Hong Kong (7005217,9220097,9220103,9229029,9229098,9678149), also by the National Natural Science Foundation of China under Grant No. 62302021.

References

[1] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, June 2021.

[2] Satanjeev Banerjee and Alon Lavie. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72, 2005.

[3] Irina Ioana Brudaru and Andreas Zeller. What is the long-term impact of changes? In *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*, page 30–32, 2008.

[4] Luis Fernando Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. On automatically generating commit messages via summarization of source code changes. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 275–284, 2014.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019*, pages 4171–4186.

[6] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature Machine Intelligence*, 5(3):220–235, 2023.

[7] Jinhao Dong, Yiling Lou, Qihao Zhu, Zeyu Sun, Zhilin Li, Wenjie Zhang, and Dan Hao. Fira: fine-grained graph-based code change representation for automated commit message generation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 970–981, 2022.

[8] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 422–431. IEEE, 2013.

[9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.

[10] Zihao Fu, Haoran Yang, Anthony Man-Cho So, Wai Lam, Lidong Bing, and Nigel Collier. On the effectiveness of parameter-efficient fine-tuning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 12799–12807, 2023.

[11] Divyam Goel, Ramansh Grover, and Fatemeh H Fard. On the cross-modal transfer from natural language to code through adapter modules. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 71–81, 2022.

[12] Yuxian Gu, Xu Han, Zhiyuan Liu, and Minlie Huang. PPT: Pre-trained prompt tuning for few-shot learning. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8410–8423, Dublin, Ireland, May 2022. Association for Computational Linguistics.

[13] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland, May 2022.

[14] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021*.

[15] Ruidan He, Linlin Liu, Hai Ye, Qingyu Tan, Bosheng Ding, Liying Cheng, Jiawei Low, Lidong Bing, and Luo Si. On the effectiveness of adapter-based tuning for pretrained language model adaptation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2208–2222, Online, August 2021. Association for Computational Linguistics.

[16] Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher A Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, et al. A fine-grained data set and analysis of tangling in bug fixing commits. *Empirical Software Engineering*, 27(6):125, 2022.

[17] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. Deepjit: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 34–45. IEEE, 2019.

[18] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 518–529, 2020.

[19] Thong Hoang, Julia Lawall, Yuan Tian, Richard J Oentaryo, and David Lo. Patchnet: Hierarchical deep learning-based stable patch identification for the linux kernel. *IEEE Transactions on Software Engineering*, 47(11):2471–2486, 2019.

[20] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International Conference on Machine Learning*, pages 2790–2799. PMLR, 2019.

[21] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

[22] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[23] Siyuan Jiang, Ameer Armaly, and Collin McMillan. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 135–146. IEEE, 2017.

[24] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2012.

[25] Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 2312–2323. IEEE Press, 2023.

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

[27] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931, 2023.

[28] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3045–3059, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[29] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, July 2020.

[30] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, Online, August 2021. Association for Computational Linguistics.

[31] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, et al. Automating code review activities by large-scale pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1035–1047, 2022.

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

[33] Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. Cct5: A code-change-oriented pre-trained model. In *Proceedings of the 31th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE, 2023.

[34] Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. Context-aware code change embedding for better patch correctness assessment. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–29, 2022.

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

[36] Zhongxin Liu, Zhijie Tang, Xin Xia, and Xiaohu Yang. Ccrep: Learning code change representations via pre-trained code model and query back. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 17–29, 2023.

[37] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

[38] Pedro Martins, Rohan Achar, and Cristina Lopes. 50k-c: a dataset of compilable, and compiled, java projects. pages 1–5, 05 2018.

[39] Chao Ni, Wei Wang, Kaiwen Yang, Xin Xia, Kui Liu, and David Lo. The best of both worlds: integrating semantic features with expert features for defect prediction and localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 672–683, 2022.

[40] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *ICLR*, 2023.

[41] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo. An empirical comparison of pre-trained models of source code. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2136–2148, Los Alamitos, CA, USA, May 2023.

[42] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.

[43] Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulić, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. AdapterHub: A framework for adapting transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 46–54, Online, October 2020. Association for Computational Linguistics.

[44] Chanathip Pornprasit and Chakkrit Kla Tantithamthavorn. Jitline: A simpler, better, faster, finer-grained just-in-time defect prediction. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 369–379. IEEE, 2021.

[45] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[46] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[47] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[48] Iman Saberi, Fatemeh Fard, and Fuxiang Chen. Utilization of pre-trained language model for adapter-based knowledge transfer in software engineering. *arXiv preprint arXiv:2307.08540*, 2023.

[49] Ensheng Shi, Yanlin Wang, Wei Tao, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. RACE: Retrieval-augmented commit message generation. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5520–5530, December 2022.

[50] Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. On the importance of building high-quality training datasets for neural code search. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1609–1620, 2022.

[51] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1433–1443.

[52] Wei Tao, Yanlin Wang, Ensheng Shi, Lun Du, Shi Han, Hongyu Zhang, Dongmei Zhang, and Wenqiang Zhang. On the evaluation of commit message generation models: An experimental study. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 126–136. IEEE, 2021.

[53] Ian Tenney, Dipanjan Das, and Ellie Pavlick. BERT rediscovers the classical NLP pipeline. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4593–4601, Florence, Italy, July 2019. Association for Computational Linguistics.

[54] Ian Tenney, Patrick Xia, Berlin Chen, Alex Wang, Adam Poliak, R Thomas McCoy, Najoung Kim, Benjamin Van Durme, Sam Bowman, Dipanjan Das, and Ellie Pavlick. What do you learn from context? probing for sentence structure in contextualized word representations. In *International Conference on Learning Representations*, 2019.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[56] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R Lyu. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 382–394, 2022.

[57] D. Wang, B. Chen, S. Li, W. Luo, S. Peng, W. Dong, and X. Liao. One adapter for all programming languages? adapter tuning for code search and summarization. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 5–16, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[58] Jiaan Wang, Fandong Meng, Duo Zheng, Yunlong Liang, Zhixu Li, Jianfeng Qu, and Jie Zhou. A survey on cross-lingual summarization. *Transactions of the Association for Computational Linguistics*, 10:1304–1323, 2022.

[59] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, November 2021.

[60] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 1482–1494. IEEE Press, 2023.

[61] Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. Commit message generation for source code changes. In *IJCAI*, 2019.

[62] Yichen Xu and Yanqiao Zhu. A survey on pretrained language models for neural code intelligence. *arXiv preprint arXiv:2212.10079*, 2022.

[63] Zhen Yang, Jacky Wai Keung, Zeyu Sun, Yunfei Zhao, Ge Li, Zhi Jin, Shuo Liu, and Yishu Li. Improving domain-specific neural code generation with few-shot meta-learning. *Information and Software Technology*, 166:107365, 2024.

[64] Zhen Yang, Jacky Wai Keung, Xiao Yu, Yan Xiao, Zhi Jin, and Jingyu Zhang. On the significance of category prediction for code-comment synchronization. *ACM Transactions on Software Engineering and Methodology*, 32(2):1–41, 2023.

[65] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 39–51, New York, NY, USA, 2022.

[66] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. Deep just-in-time defect prediction: how far are we? In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 427–438, 2021.

[67] Ningyu Zhang, Luoqiu Li, Xiang Chen, Shumin Deng, Zhen Bi, Chuanqi Tan, Fei Huang, and Huajun Chen. Differentiable prompt makes pre-trained language models better few-shot learners. In *International Conference on Learning Representations*, 2022.

[68] Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474*, 2022.