Parameter-Efficient Fine-Tuning of Large Language Models for Unit Test Generation: An Empirical Study

1st André Storhaug
Department of Computer Science
Norwegian University of Science and Technology
Trondheim, Norway
andre.storhaug@ntnu.no

2nd Jingyue Li
Department of Computer Science
Norwegian University of Science and Technology
Trondheim, Norway
jingyue.li@ntnu.no

***Abstract*—The advent of large language models (LLMs) like GitHub Copilot has significantly enhanced programmers' productivity, particularly in code generation. However, these models often struggle with real-world tasks without fine-tuning. As LLMs grow larger and more performant, fine-tuning for specialized tasks becomes increasingly expensive. Parameter-efficient fine-tuning (PEFT) methods, which fine-tune only a subset of model parameters, offer a promising solution by reducing the computational costs of tuning LLMs while maintaining their performance. Existing studies have explored using PEFT and LLMs for various code-related tasks and found that the effectiveness of PEFT techniques is task-dependent. The application of PEFT techniques in unit test generation remains underexplored. The state-of-the-art is limited to using LLMs with full fine-tuning to generate unit tests. This paper investigates both full fine-tuning and various PEFT methods, including LoRA, (IA)^3, and prompt tuning, across different model architectures and sizes. We use well-established benchmark datasets to evaluate their effectiveness in unit test generation. Our findings show that PEFT methods can deliver performance comparable to full fine-tuning for unit test generation, making specialized fine-tuning more accessible and cost-effective. Notably, prompt tuning is the most effective in terms of cost and resource utilization, while LoRA approaches the effectiveness of full fine-tuning in several cases.**

***Index Terms*—large language models, parameter-efficient fine-tuning, test generation, unit tests**

I. Introduction

Large language models (LLMs) have been developed for various tasks, including general language and code-related activities, significantly enhancing programmers' performance. Training LLMs on extensive code bases like GitHub give them seemingly magical capabilities. However, despite their proficiency in multiple code-related tasks, LLMs still struggle with cost-effectiveness in their training. The traditional approach to handling these specialized tasks has been fine-tuning the model specifically for each task. Given that LLMs often have billions, if not trillions, of parameters, traditional fine-tuning, which adjusts all parameters, is extremely computationally expensive and resource-intensive. However, state-of-the-art approaches [1] still rely on energy-costly full fine-tuning.

Advancements in instruction-tuned LLMs have encouraged investigations into various prompting and tuning techniques, e.g., [2, 3, 4], that do not require fine-tuning. This challenge has led to the development of multiple techniques for fine-tuning larger models with limited resources, commonly referred to as parameter-efficient fine-tuning (PEFT).

The application of various PEFT methods has been explored for code-related generation tasks, such as code completion, code summarization, and more [5, 6, 7, 8, 9, 10, 11, 12, 13]. Evaluations of the PEFT methods show that they could achieve performance comparable to full fine-tuning while significantly reducing the computational burden by strategically fine-tuning only a chosen subset of parameters during the code-related generation process. However, the effectiveness of PEFT techniques on different tasks varies.

LLMs have been used to generate unit test cases and assertion statements [14, 12, 2, 3, 4, 1]. However, to the best of our knowledge, the area of unit test generation using PEFT remains largely unexplored. Therefore, in this work, we empirically evaluate the performance of various PEFT methods for unit test generation. We focus on answering the following research questions:

- **RQ1: How well does PEFT perform on unit test generation?**
- **RQ2: What is the relation between resource utilization and performance of PEFT in unit test generation?**

To answer the research questions, we compared full fine-tuning with three popular PEFT methods: LoRA (Low-Rank Adaptation) [15], (IA)^3 (Infused Adapter by Inhibiting and Amplifying Inner Activations) [16], and prompt tuning [17]. We conducted these comparisons using ten LLMs from three open-source, decoder-only LLM families. The LLMs have varying architectures and sizes, ranging from 350 million to 16 billion parameters. We measured and compared the unit test generation performance of full fine-tuning and PEFT methods using well-established benchmark datasets rooted in real-world unit test cases.

Our results show that LoRA is generally the most reliable method for improving the quality of the generated unit tests and is, in many cases, on par with or better than full fine-tuning. Conversely, (IA)^3 appears to be the least effective, often resulting in minimal improvements while consistently retaining prior knowledge in the LLMs. Prompt tuning demonstrates significant variability, excelling with some models, particularly larger ones, while underperforming with others. This variability highlights the importance of considering model size and characteristics when choosing the appropriate tuning method. Prompt tuning can be considered ”low-hanging fruit,” while LoRA should be employed for greater stability. Our contributions are as follows:

- We conducted the first empirical study to extensively evaluate training LLMs using various PEFT methods—LoRA, (IA)^3, and prompt tuning—for unit test generation across a wide range of LLM models.
- We offered a comprehensive comparison of PEFT methods versus full fine-tuning.
- Our findings yield practical guidelines highlighting the potential gains and limitations of the various PEFT methods and full fine-tuning in resource utilization and performance.

The rest of the paper is organized as follows. We provide preliminaries in Section II. Section III introduces related work. We explain our experimental design in Section IV. Section V presents the experimental results. Section VI discusses our results and limitations. Section VII concludes the study and proposes future work.

II. Preliminaries

A. LLM training

The classic language modeling task involves predicting the next word in a sequence, given the preceding words. The model is trained in an autoregressive manner. A typical example is the GPT model [18]. Training an autoregressive model involves fitting a function $f : \mathbb{R}^{N\times D} \to \mathbb{R}^{N\times D}$, where N represents the sequence length of tokens, each token having a dimension of D. The model aims to estimate the conditional probability distribution, denoted p, which can be expressed as:

$$p(x_{1:T} \mid c) = \prod_{t=1}^{T} p(x_t \mid x_{1:t-1}, c) \tag{1}$$

where $x_t \in \mathbb{R}^D$ represents the t'th discrete token observation, c denotes the initial context or input sequence, and T signifies the length of the sequence of predicted tokens [19, Chapter 22].

LLMs are normally pre-trained on a huge amount of data, giving them a broad understanding of language patterns and semantics. This pre-training phase allows LLMs to capture intricate linguistic structures and contextual nuances, empowering them with a robust foundation for various natural language processing tasks.

Fine-tuning is a method in which a pre-trained model can be adapted to a more specific task, such as auto-completing source code. The process involves adjusting the weights of the pre-trained model using task-specific data, thereby enhancing its performance on a target task. When *all* parameters are tuned, this is often referred to as “full fine-tuning” [20]. If we imagine the model as a web of knowledge, the full fine-tuning loosens all the connections and lets them adapt entirely to the new domain. This flexibility allows for potentially superior performance but comes at a cost:

- Overfitting: The model can become overly reliant on the training data, potentially struggling with unseen examples [21].
- Computational cost: Training all parameters requires significant computational resources [20].
- Catastrophic forgetting: The model might forget some valuable knowledge it acquired during pre-training [22, 23].

B. PEFT Methods

Parameter-efficient fine-tuning (PEFT) methods provide an alternative way of fine-tuning a pre-trained model by training so-called adapters [24]. Here, most (if not all) parameters of the pre-trained model are frozen, and only a small number of parameters are trained. These adapters are often orders of magnitude smaller than the full model, making sharing, storing, and loading especially convenient. Moreover, some PEFT methods even facilitate the utilization of mixed-task batches, enabling distinct processing for different examples within a batch [17].

PEFT methods can be classified into three main classes of methods: additive, selective, and reparametrization-based [25]. Additive methods can be further broken down into adapters [1] and soft prompts.

1) Additive: This approach focuses on augmenting the pre-trained model with additional parameters or layers. Only the newly added components are trained, keeping the original model weights frozen. Popular techniques within the additive category are (IA)^3 [16], adapters [26], and prompt tuning [17].

a) Adapters: Adapter, or “bottleneck adapter,” introduced by Houlsby et al. [26] was arguably the first PEFT method. The class of “adapters” introduces small, fully connected networks placed after specific sub-layers within the model. These help the model adapt to the new task without significantly altering the core representation learned during pre-training.

b) Soft prompts: These techniques focus on creating the so-called “soft prompts.” Prompt tuning, introduced by Lester et al. [17], fine-tuned a subset of the model’s input embeddings, allowing the model to adapt to new tasks without altering the model’s parameters. The prefix tuning expands the idea to all model layers [27].

2) Selective: When using selective approaches, only a specific subset of the pre-trained model’s parameters is fine-tuned. This approach requires carefully selecting the parameters to be updated, ensuring that they are most relevant to the new task. An example is BitFit by Ben Zaken et al. [28], which only updates the bias parameters.

[1]The term ”adapter” is also used as the name of the first proposed adapter module for NLP.

3) Reparametrization-based: This category utilizes techniques that reformulate the original model's parameters into a more efficient trainable representation. This makes it possible to work with high-dimensional matrices while simultaneously reducing the number of trainable parameters. LoRA [15] is a common reparametrization-based method.

III. Related Work

This section presents state-of-the-art studies generating unit tests using LLMs and PEFT approaches to enhance code-related tasks.

A. Unit test generation using LLMs

Researchers have turned to neural techniques to automatically generate unit tests to align with real-world best practices in unit test generation. Tufano et al. [1] propose an automated unit test case generation tool called AthenaTest. They construct a real-world focal method and developer-written test case dataset named Methods2Test [2] and train a sequence-to-sequence transformer to generate unit test cases.

Advancements in instruction-tuned LLMs have also encouraged several investigations of various prompting techniques, as these do not require any fine-tuning [2, 3, 4]. Yuan et al. [3] propose ChatTester, a ChatGPT-based unit test generation approach, which leverages ChatGPT itself to improve the quality of its generated tests. Schäfer et al. [4] conducted a large-scale empirical evaluation on the effectiveness of LLMs for automated unit test generation without additional training or manual effort. Similarly to ChatTester, they also use ChatGPT (GPT-3.5-Turbo) and propose a re-prompting approach named TestPilot that automatically generates unit tests for all API functions in an npm package. Siddiq et al. [2] use the HumanEval and SF110 datasets to benchmark how well Codex, GPT-3.5-Turbo, and StarCoder can automatically generate unit tests. They evaluate the models based on compilation rates, test correctness, test coverage, and test smells.

B. PEFT methods on code-related tasks

While prompting methods are gaining traction, they often require significant manual effort to design complex templates and rely on expensive enterprise-grade LLMs like ChatGPT. Consequently, most cutting-edge methods for automatic code generation still involve computationally expensive full fine-tuning. One alternative approach to full fine-tuning is proposed by Steenhoek et al. [29]. They use Reinforcement Learning from Static Quality Metrics (RLSQM). As reinforcement learning is a very expensive fine-tuning method, they also see the value of using PEFT methods like LoRA to facilitate the training.

PEFT has emerged as a powerful technique for enhancing code generation with LLMs. Researchers are investigating how these models perform on code tasks using PEFT techniques as a more efficient alternative to full fine-tuning and PEFT techniques' effectiveness for different code-related tasks. Results show that the effectiveness of PEFT techniques is task-dependent and not transferrable. For example, Ayupov and Chirkova [5] evaluated adapters and LoRA on various tasks, finding them effective for code understanding but less so for code generation compared to full fine-tuning. Wang et al. [7] reports that prompt tuning of CodeBERT and CodeT5 outperforms full fine-tuning for defect prediction, code summarization, and code translation tasks. Liu et al. [13] evaluated adapter, LoRA, Prefix tuning, and Mix-And-Match adapter (MAM) [30] for defect detection, code clone detection, and code summarization. They concluded that "*PEFT may achieve comparable or higher performance than full fine-tuning in code understanding tasks, but they may exhibit slightly weaker performance in code generation tasks.*" Similarly, Weyssow et al. [8] evaluate text-to-code generation using LoRA, (IA)^3, prompt tuning, and prefix tuning. They compare performance across models of varying sizes (up to 7 billion parameters) against full fine-tuning of smaller models (less than 350 million parameters) and concluded that LoRA is the most effective one.

To our knowledge, there is currently no systematic knowledge of how various PEFT methods behave in the domain of unit test generation. As existing studies show that PEFT techniques' effectiveness is task-dependent, it is, therefore, necessary to empirically evaluate their effectiveness for unit test generation.

IV. Experimental design

The main objective of this study is to gain insight into the performance of fine-tuning LLMs using PEFT for unit test generation.

A. Research Questions

To address RQ1 ("**How well does PEFT perform on unit test generation?**"), we evaluate several different PEFT methods on unit test generation. We also compare the PEFT methods with full fine-tuning. Our study covers a wide range of language models for code of various sizes, spanning 350 million to 16 billion parameters. We assess the quality of the generated test cases produced by different PEFT methods by scoring against real-world human-written reference tests and evaluating the test cases' syntactic correctness. Further, we examine whether applying PEFT methods for unit test generation adversely affects the pre-trained models' code generation performance, also known as catastrophic forgetting [22, 23]. This evaluation will help us understand if there are any trade-offs between applying fine-tuning and PEFT methods for unit test generation and maintaining performance on other code-related tasks.

Regarding RQ2 ("**What is the relation between resource utilization and performance of PEFT in unit test generation?**"), we want to understand PEFT methods' performance gains in relation to their resource utilization and requirements. Thus, we analyze the efficiency of PEFT methods in terms of

[2]https://github.com/microsoft/methods2test

the computational resources they use and their impact on the performance outcomes in unit test generation.

B. Fine-tuning methods selection

In line with other empirical studies on PEFT [8, 31], we select the popularly used PEFT methods: LoRA [15], (IA)^3 [16], and prompt tuning [17]. They are readily available in popular libraries and exemplify most of the primary approaches within PEFT.

- **LoRA**: A reparametrization-based method that updates a weight matrix by training a new matrix that is decomposed into a product of two matrices of lower rank: $\delta W = W_A \cdot W_B$, where $W_A \in \mathbb{R}^{\text{in} \times r}$ and $W_B \in \mathbb{R}^{r \times \text{out}}$ [15]. As demonstrated in Figure 1a, the decomposed matrix is then added to the original frozen K and V projection matrices in the attention modules. All other modules are kept frozen.
- **(IA)^3**: An additive method that focuses on fine-tuning specific intermediate activations within the model [16]. As illustrated in Figure 1b, this is done by learning three new vectors: l_v, l_k, l_{ff}; rescaling key, value, and hidden feed-forward activations, respectively.
- **Prompt tuning**: A soft prompts method that fine-tunes a subset of the model's input embeddings. Exemplified in Figure 1c, the model input embeddings are prepended with a trainable tensor $P \in \mathbb{R}^{N \times D}$ (known as a "soft prompt") [17]. Each element of P is also referred to as a "virtual token." During training, these "virtual tokens" are updated while all the rest of the model parameters are kept frozen.

C. Models

For our experiments, we select a diverse set of open-source decoder-only model families trained on code to evaluate the performance and efficiency of PEFT methods. The chosen models span a range of sizes and architectures and are trained on diverse datasets and training objectives. This offers extensive insights into the interplay between various model capacities and PEFT methods. Further, all models are ensured to support the Java programming language to facilitate a fair comparison of our benchmarking datasets described in Section IV-D.

- **Salesforce CodeGen Models**: To democratize program synthesis, Salesforce AI Research developed CodeGen [32]. Pioneering in multi-turn program synthesis, CodeGen presents a variety of sizes, ranging from 350 million to 16 billion parameters. These models have undergone training across a diverse array of both programming and natural languages. The models were trained on The pile [33], followed by a code subset of Google's BigQuery datasets Cloud [34]. An updated version CodeGen2 [35] is also available in sizes 1 billion to 16 billion, trained on The Stack dataset [36] using both causal language modeling (CLM) and file-level span corruption. We select the smallest model of the first generation CodeGen family (*CodeGen-350M-multi*), along with all model sizes of the second generation (1 billion, 3.7 billion, 7 billion, and 16 billion parameters).
- **Meta LLaMA Models**: Code Llama [37], derived from Llama 2 [38], is a specialized model further fine-tuned on the code-specific parts of its pre-training data. With enhanced coding capabilities, it can generate code, interpret natural language prompts, assist in code completion, and aid debugging. It comes in three variants. One is optimized for instructions. Another for Python. We select the base variant *CodeLlama-7B*, as it is designed for general code synthesis and understanding and supports a wide range of popular languages, including Java.
- **BigCode StarCoder Models**: This model family includes the 15.5 billion parameters *StarCoderBase* model [39], along with an updated model generation named StarCoder2 [40], available in 3 billion, 7 billion, and 15 billion parameters. StarCoderBase was trained on 80+ programming languages from The Stack dataset [36]. The updated StarCoder2 is trained on 619 programming languages and other data sources, such as GitHub pull requests, Kaggle notebooks, and code docs. We select all model generations and sizes in this model family.

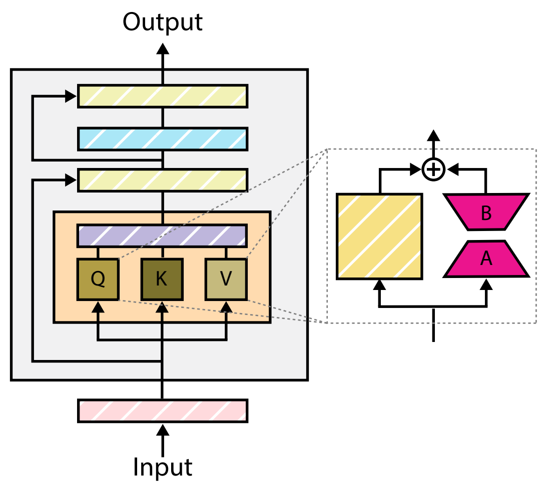
D. Datasets

1) METHODS2TEST*:* To ensure our experiments' validity and practical applicability, we rely on METHODS2TEST [1]. It is the largest publicly available corpus of Java unit test cases and corresponding focal methods. This dataset contains 780,000 test cases sourced from 91,000 open-source repositories on GitHub. These real-world test cases represent the complexity and variability inherent in software development, making them an ideal choice for evaluating automated test generation methods. By leveraging METHODS2TEST, we can assess the performance of the various PEFT methods and models in generating high-quality unit tests that align with real-world coding practices and challenges.

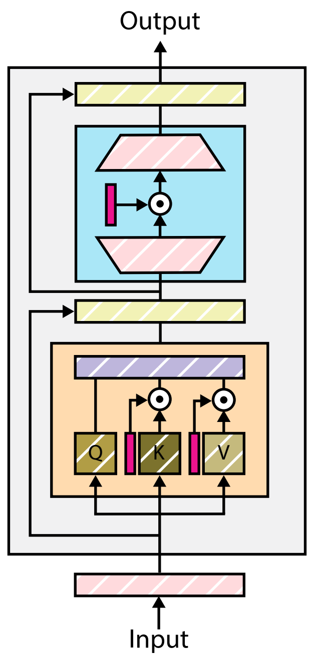
One of the advantages promised by various PEFT methods is the reduction in the amount of data needed for effective training. Thus, we opt to downsize the METHODS2TEST dataset to enhance efficiency without compromising the representativeness of the data. Specifically, we adopt a strategy wherein only one focal function is selected from each class in the dataset. By doing so, we maintain the essence and diversity of the original dataset while significantly reducing its size. Further, the METHODS2TEST dataset is distributed in deconstructed components. Hence, we assemble it according to the structure seen in Listing I and repackage it as METHODS2TEST$_{\text{SMALL}}$[3]. The resulting dataset contains 7,44K training, 953 validation, and 1.02K testing tests.

2) HumanEval-X: HumanEval is a benchmarking dataset for code generation, first introduced by Chen et al. [41]. It consists of 164 hand-written programming problems written in Python. For each programming problem, a function signature, docstring, body, and several unit tests are provided. Zheng

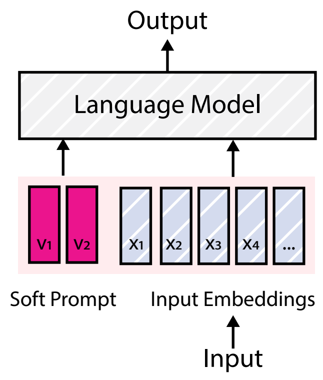
[3]Available at https://huggingface.co/datasets/andstor/methods2test_small



(a) Diagram of LoRA (based on [15]).



(b) Diagram of $(IA)^3$ (based on [16]).



(c) Diagram of prompt tuning (based on [17]).

Fig. 1: PEFT methods architecture illustration diagrams based on the reference implementation of each introductory paper. Striped modules are frozen. Magenta modules are trained.

```
// Focal class
public class Calculator {

    // Focal method
    public float add(float value) {
        this.previousValue = this.currentValue;
        this.currentValue += value;                  } fm  } +fc  } +c  } +m  } +f
        return this.currentValue;
    }

    // Constructor signatures
    public Calculator();
    public Calculator(float value);

    // Method signatures
    public float subtract(float value);
    public float multiply(float value);
    public float divide(float value);
    public void reset();

    // Fields
    public float currentValue;
    public float previousValue;
}

// Test class
public class CalculatorTest {

    // Test case
    @Test
    public void testAdd() {
                                                                      Source
                                                                      Target
        Calculator calc = new Calculator();          } t  } +tc
        assertEquals(5.0, calc.add(5.0f), 0.001);
        assertEquals(7.0, calc.add(2.0f), 0.001);
    }
}
```

LISTING I: Dataset components: focal method (*fm*), focal class name (*fc*), constructor signatures (*c*), method signatures, fields (*f*), test case (*t*), test class name (*tc*). Data is assembeld according to *fm*+*fc*+*c*+*m*+*f*+*t*+*tc*. The code is split after the test case signature. Everything above the red line is the *source* input for unit test generation. The code below the red line is the *target* output of the unit test generation.

et al. [42] extended the dataset to multiple languages, named HumanEval-X. We use the Java subset of the HumanEval-X dataset to evaluate potential adverse fine-tuning effects.

E. Training

As the training dataset METHODS2TEST$_{\text{SMALL}}$ only includes method signatures for non-focal methods, the unit test generation task can be considered a sequence-to-sequence (seq2seq) problem. For training the decoder-only models described in Section IV-C, we, therefore, model the seq2seq task as an autoregressive problem. This is done by only calculating loss based on the *target* code, i.e., the unit test code. We minimize the following autoregressive cross-entropy loss function:

$$\mathcal{L} = -\sum_{t=1}^{T} \mathbb{1}_t \cdot \log p(x_t \mid x_{1:t-1}), \tag{2}$$

where:

$$\mathbb{1}_i = \begin{cases} 1, & \text{if } x_i \neq -100 \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

The model is fed the *source* and *target* concatenated together (see Listing I). The token values for the *source* are set to -100. The indicator function $\mathbb{1}_i$ is used to ignore the *source* in the loss computation.

We use the Transformers [43] and PEFT libraries [44] from Hugging Face to conduct all training. We use NVIDIA A100 GPUs and leverage DeepSpeed ZeRO-Offload [45] where necessary. We use the hyperparameters shown in Table I for

TABLE I: Model-agnostic hyperparameters for training.

Hyperparameter	Method	Value
Common		
Optimizer	-	AdamW
LR schedule	-	Linear
LR warmup ratio	-	0.1
Batch size	-	1
Gradient accumulation steps	-	8
# Epochs	-	3
Precision	-	Mixed
Learning rate	Full fine-tuning	5E-5
	LoRA	3E-4
	$(IA)^3$	3E-4
	Prompt tuning	3E-3
Method specific		
Alpha	LoRA	32
Dropout	LoRA	0.1
Rank	LoRA	16
Virtual tokens	Prompt tuning	20

training. Like other studies, e.g., [8], we select hyperparameter values that align with those specified in the original papers of the respective PEFT techniques. Weyssow et al. [8] fine-tuned their models for up to five epochs. Our results show that the losses of most of our fine-tuned models are stable after three epochs[4]. Regarding the unspecified hyperparameters, we leave them to library defaults. Following defaults of the PEFT library, rank decomposition for LoRA is applied to the attention layers. The same attention layers and the default feed-forward modules (see Table II) are targeted for $(IA)^3$. The selected models have context windows ranging from 2048 to 16,384. To make the experiments align with each other, we limit the number of tokens to the smallest window of 2048.

F. Generation

For generating, we provide the *source* part of the data (see Listing I) as input to the models, while the *target* part (original code) is used as the ground truth. The generated unit test code is then compared to the ground truth to calculate their similarities. We implement a stopping strategy based on matching braces to generate well-formed functions. For incomplete unit tests, we follow the protocol from [2] where we iteratively delete lines (from bottom to top), add one to two curly brackets, and check if the syntax is valid using an ANTLR v4-based [46] Java parser [47]. We only evaluate samples that are well-formed.

We also apply DeepSpeed-Inference [48] to facilitate model parallel (MP) to fit the larger models that would otherwise not fit in GPU memory. Table III shows the hyperparameter details used for generation.

[4]Available at https://huggingface.co/andstor/peft-unit-test-generation-experiments

G. Metrics

1) Syntactic Correctness: Following Yuan et al. [3], we measure the syntactic correctness of the generated unit tests, ensuring that they conform to the grammar of the Java programming language. After the fixing described in Section IV-F, we employ the same parser as described in Section IV-F. This involves parsing the generated tests and verifying that they are free of syntax errors. We then report the percentage of unit tests with valid syntax for each model.

2) Similarity: BLEU score [49] is a very common metric used to evaluate natural language machine translation. It measures the similarity of the machine-translated text to the reference translations. However, Ren et al. [50] recognized that it neglects important syntactic and semantic features of programming languages. To remedy this, they developed CodeBLEU [50]. Based on n-gram matching used in BLEU, they further augment the scoring by leveraging abstract syntax trees (AST) and code semantics via data flow. To evaluate the primary performance of the various PEFT methods, we use CodeBLEU to measure the similarity between the generated unit test code and the ground truth.

3) Coverage: We also execute the generated tests for the HUMANEVAL-X$_{\text{JAVA}}$, and calculate code coverage. Specifically, we calculate the pass@1 metric [41] and instruction and branch coverage using the Java Code Coverage Library JaCoCo [51]. Instruction coverage measures the percentage of executed instructions, while branch coverage evaluates the percentage of executed control flow branches in the program.

V. EXPERIMENTAL RESULTS

This section presents the detailed results of each research question.

A. RQ1: Tuning method performance

We present the performance results of the unit test generation for the various models and tuning methods in Table IV.

1) Syntactic Correctness: As shown in Table IV, the percentage of syntactically correctly generated unit tests is relatively high across the board, with $> 80\%$ for all except the smaller CodeGen2-1B and CodeGen2-3.7B models. These two cannot produce any valid unit test with the base model. However, after tuning, both models are able to produce up to 76.73% and 52.24% syntactically valid tests, respectively. The notable findings (marked with underlines in Table IV) are as follows: 1) The increase of 66.63% using prompt tuning of CodeGen2-1B, tuning only 0.02‰ (0.04 million out of 1 billion parameters) of the parameters. 2) The increase of 52.24% using LoRA on CodeGen2-3.7B, tuning only 0.1% (4.19 million out of 3.7 billion parameters).

Further, data in Table IV illustrate that: 1) Full fine-tuning achieves higher percentages of syntactically correct unit tests than the baseline. The median percentage of syntactically valid tests for the full fine-tuning is 97.01%, which is 13.69% higher than the value of the baseline (85.33%). 2) There is a weak tendency that the more trainable parameters, the more samples are generated syntactically correctly. The median percentage

TABLE II: Model-specific hyperparameters for training.

Hyperparameter	Method	Model	Value
Targeted attention modules	LoRA, (IA)^3	codegen-350M-multi	qkv_proj
		Salesforce/codegen2-1B_P	qkv_proj
		Salesforce/codegen2-3_7B_P	qkv_proj
		Salesforce/codegen2-7B_P	qkv_proj
		Salesforce/codegen2-16B_P	qkv_proj
		meta-llama/CodeLlama-7b-hf	q_proj, v_proj
		bigcode/starcoderbase	c_attn
		bigcode/starcoder2-3b	q_proj, v_proj
		bigcode/starcoder2-7b	q_proj, v_proj
		bigcode/starcoder2-15b	q_proj, v_proj
Targeted feedforward modules	(IA)^3	codegen-350M-multi	fc_out
		Salesforce/codegen2-1B_P	fc_out
		Salesforce/codegen2-3_7B_P	fc_out
		Salesforce/codegen2-7B_P	fc_out
		Salesforce/codegen2-16B_P	fc_out
		meta-llama/CodeLlama-7b-hf	down_proj
		bigcode/starcoderbase	mlp.c_proj
		bigcode/starcoder2-3b	q_proj, c_proj
		bigcode/starcoder2-7b	q_proj, c_proj
		bigcode/starcoder2-15b	q_proj, c_proj

TABLE III: Hyperparameters for generation.

Hyperparameters	Value
Do sample	False
Temperature	0
Top p	0
Frequency penalty	0
Max length	2048

of syntactically valid tests for the full fine-tuning is 97.01%, which is slightly higher than LoRA (96.96%), (IA)^3 (91.13%), and prompt tuning (91.18%). However, considering that full fine-tuning often tunes parameters several thousands more times than PEFT methods, it is probably not wise to use full fine-tuning to improve the syntactic correctness of the unit tests if cost-effectiveness is a priority.

2) Similarity: The METHODS2TEST$_{\text{SMALL}}$ dataset CodeBLEU column in Table IV shows the CodeBLEU scores of the baseline, full fine-tuning, and PEFT methods. The notable results are: 1) LoRA archives the highest CodeBLEU scores in five of the ten models across different sizes. LoRA is also the only method that can positively increase the CodeBLEU score of the CodeLlama-7B model. 2) Full fine-tuning is nearly as performant as LoRA. For three of the ten models, full fine-tuning scores the best. 3) (IA)^3 performs generally poorly compared to others, often resulting in very low CodeBLEU scores. 4) Prompt tuning shows varying results, from being the worst performing on CodeLlama-7B to being the best performing on CodeGen2-1B.

3) Coverage: From the pass@1 column in Table IV, the average mean pass@1 rate is 0.2332. Of the tests that pass, the code coverage is generally high, as can be seen from the two last columns in Table IV. The exception is where the model is not able to generate much syntactically valid code. This results in some coverage results being zero, e.g. CodeGen2-1B and CodeGen2-3.7B. With an average median instruction coverage of 98.81% and branch coverage of 78.55%, this shows that if the models are able to create a runnable test, they are likely to cover a significant portion of the code. This high level of coverage indicates that the generated tests are effective at exercising the codebase, potentially identifying edge cases, and ensuring robustness. The results also show that LoRA and full fine-tuning outperform other methods in several cases. Although (IA)^3 and prompt tuning perform best in some cases, the differences between their performance and others are insignificant.

4) Resilience against catastrophic forgetting: The results in the last five columns of Table IV show the evaluation of various PEFT tuning methods on the Java subset of HumanEval-X after they have been trained on the METHODS2TEST$_{\text{SMALL}}$ dataset. As expected, we can see from the results that there are some cases of degradation (marked using parentheses in Table IV) when comparing the results to the baseline. This applies to both full fine-tuning and PEFT. However, the levels of degradation are, for the most part, trivial. We also see several cases, e.g., StarCoder2-15B, where PEFT tuning actually improves the CodeBLEU scores. This shows that neither tuning method is very susceptible to catastrophic forgetting, and PEFT methods might pose similarly good transfer learning capabilities to that of full-fine-tuning.

Summary for RQ1: Full fine-tuning and PEFT methods could improve the syntactic correctness of the generated unit tests. PEFT methods could be better choices than full fine-tuning if cost-effectiveness to improve the syntactic correctness of the generated unit tests is a concern. Full fine-tuning and LoRA could significantly boost the quality of the generated unit tests measured using CodeBLU, with LoRA even surpassing full fine-tuning in several cases. Further, PEFT methods are mostly resilient against catastrophic forgetting.

TABLE IV: Comparison of syntactical validity and CodeBLEU scores from experiments using different tuning methods across various models on testing split of METHODS2TEST$_{\text{SMALL}}$ and HUMANEVAL-X$_{\text{JAVA}}$ datasets.

Model	**Method**	**Trainable params**	**METHODS2TEST$_{\text{SMALL}}$**		**HUMANEVAL-X$_{\text{JAVA}}$**				
			Valid syntax	CodeBLEU	Valid syntax	CodeBLEU	pass@1	Instr Cov	Branch Cov
CodeGen-350M-multi	None	0	95.43%	0.2170	100%	0.3608	0.0671	97.33%	89.77%
	Full fine-tuning	304.23M	97.87%	**0.2988**	100%	(0.3293)	(0.0366)	100%	(83.33%)
	LoRA	1.31M	(95.22%)	0.2553	100%	**0.3907**	**0.0671**	**98.69%**	(89.29%)
	(IA)^3	0.14M	95.53%	0.2266	100%	(0.3583)	(0.0549)	97.69%	**94.32%**
	Prompt tuning	0.02M	96.03%	0.2208	100%	(0.3290)	(0.0427)	(96.56%)	91.67%
CodeGen2-1B	None	0	0%	0	0%	0	0	0%	0%
	Full fine-tuning	1,015.31M	76.73%	0.1474	5.49%	0.0359	0	0%	0%
	LoRA	2.10M	41.16%	0.0484	8.54%	0.0117	0	0%	0%
	(IA)^3	0.23M	1.52%	0.2553	0%	0	0	0%	0%
	Prompt tuning	0.04M	66.63%	**0.2568**	7.93%	**0.2547**	0	0%	0%
StarCoder2-3B	None	0	85.23%	0.1543	100%	0.4264	0.3152	9.89%	85.67%
	Full fine-tuning	3,030.37M	96.71%	0.2786	100%	**0.4969**	0.3494	99.40%	**86.37%**
	LoRA	4.55M	97.11%	**0.2901**	100%	(0.4169)	**0.3675**	99.19%	58.84%
	(IA)^3	0.47M	87.43%	0.2513	100%	(0.4250)	(0.2744)	**99.67%**	(83.81%)
	Prompt tuning	0.06M	86.43%	0.1742	100%	0.4309	(0.2470)	99.6%	(75.85%)
CodeGen2-3.7B	None	0	0%	0	0%	0	0	0%	0%
	Full fine-tuning	3,641.17M	50.51%	0.1006	73.78%	0.2621	0	0%	0%
	LoRA	4.19M	52.24%	0.0997	40.24%	**0.1384**	0	0%	0%
	(IA)^3	0.46M	0%	0	0%	0	0	0%	0%
	Prompt tuning	0.08M	23.50%	**0.2562**	0%	0	0	0%	0%
CodeLlama-7B	None	0	97.66%	0.3107	99.39%	0.4861	0.3293	98.33%	84.46%
	Full fine-tuning	6,607.41M	(96.44%)	(0.3012)	100%	**0.4994**	**0.3373**	98.95%	**86.37%**
	LoRA	8.39M	(97.36%)	**0.3277**	99.39%	(0.4291)	(0.3129)	**99.61%**	(72.47%)
	(IA)^3	0.61M	(97.05%)	(0.3011)	100%	(0.4802)	(0.3232)	98.77%	84.72%
	Prompt tuning	0.08M	(95.93%)	(0.2885)	99.39%	(0.4617)	(0.2761)	98.38%	(82.25%)
CodeGen2-7B	None	0	96.95%	0.2848	100%	0.4736	**0.2256**	98.31%	**81.45%**
	Full fine-tuning	6,862.87M	97.56%	0.3107	100%	(0.4398)	(0.1280)	**99.75%**	(70.00%)
	LoRA	8.39M	97.87%	**0.3164**	100%	(0.4636)	(0.2073)	(98.06%)	(75.35%)
	(IA)^3	0.92M	97.36%	0.2904	100%	**0.4898**	(0.1829)	98.55%	(79.50%)
	Prompt tuning	0.08M	(96.64%)	(0.2775)	100%	(0.4407)	(0.2012)	99.10%	(69.40%)
StarCoder2-7B	None	0	84.13%	0.1610	100%	0.4027	**0.3758**	99.07%	83.16%
	Full fine-tuning	7,173.92M	97.21%	0.3009	100%	0.4389	(0.3675)	99.15%	**90.80%**
	LoRA	7.34M	96.91%	**0.3068**	100%	**0.5179**	(0.3394)	99.35%	87.06%
	(IA)^3	0.75M	94.83%	0.2903	100%	0.4213	(0.3697)	**99.40%**	88.39%
	Prompt tuning	0.09M	(83.03%)	0.3030	100%	0.5057	(0.3476)	99.38%	86.02%
StarCoderBase	None	0	84.63%	0.1563	98.78%	0.4338	0.2963	99.07%	81.48%
	Full fine-tuning	15,517.46M	96.81%	0.3123	100%	**0.4830**	**0.3293**	99.16%	(75.20%)
	LoRA	8.03M	95.71%	**0.3152**	98.78%	(0.3905)	0.2963	99.07%	(73.89%)
	(IA)^3	1.24M	84.63%	(0.1553)	98.78%	0.4344	(0.1562)	(98.72%)	**81.48%**
	Prompt tuning	0.12M	85.73%	(0.1518)	(78.05%)	(0.2315)	0.3025	**99.76%**	(67.62%)
StarCoder2-15B	None	0	85.43%	0.1898	100%	0.3724	0.4085	98.93%	87.69%
	Full fine-tuning	15,655.90M	97.90%	**0.3323**	(99.39%)	0.4886	0.3758	99.52%	81.3%
	LoRA	12.12M	97.01%	0.3272	100%	0.4633	0.4146	98.95%	82.88%
	(IA)^3	1.25M	85.43%	0.1901	100%	0.3725	**0.4578**	**99.57%**	**87.89%**
	Prompt tuning	0.12M	97.60%	0.3133	100%	**0.5352**	0.3939	99.32 %	82.89 %
CodeGen2-16B	None	0	97.87%	0.2784	100%	0.4779	0.2012	98.66%	80.56%
	Full fine-tuning	16,032.16M	(97.56%)	**0.3383**	(98.17%)	(0.3774)	(0.1180)	**99.52%**	(78.07%)
	LoRA	13.37M	98.68%	0.3186	100%	(0.4714)	0.2012	98.66%	**82.06%**
	(IA)^3	1.46M	97.87%	0.2790	100%	**0.4780**	0.2134	(97.46%)	80.56%
	Prompt tuning	0.08M	97.97%	0.2954	100%	(0.4679)	**0.2195**	(98.62%)	(71.07%)

Bold: best-performing training method per model. (Parentheses): decreased performance compared to baseline. Red : $<$ 50% syntactical valid samples. Underline: Other notable results (see in Section V-A1).

B. RQ2: PEFT training resource utilization effectiveness

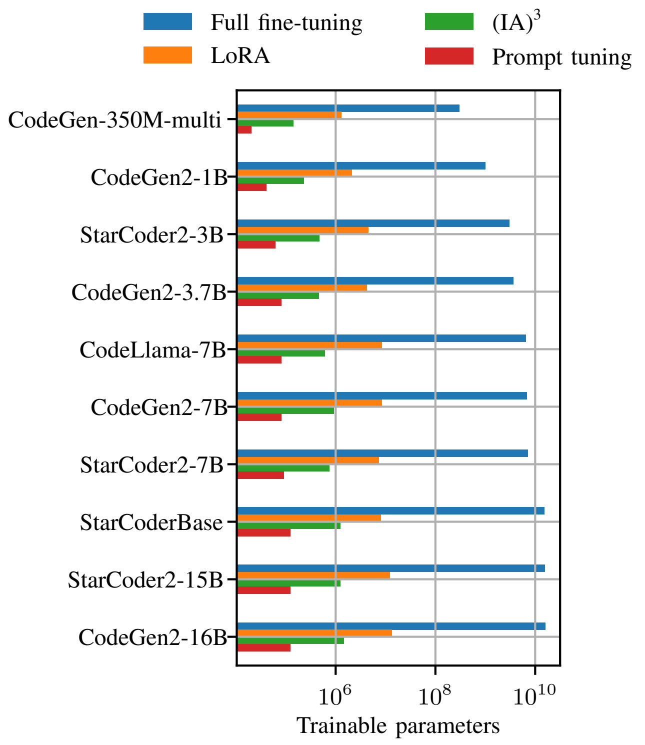


Fig. 2: Comparison of the number of trainable parameters for each tuning method. Models are sorted by size.

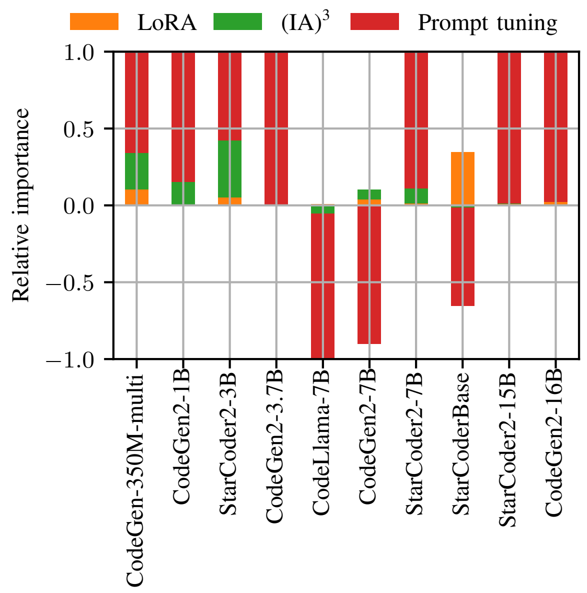


Fig. 3: Stacked bar chart of symmetrically normalized CodeBLEU score increase and decrease, relative to the fraction of trained parameters for each model. Positive values mean an increased CodeBLEU score, while negative values indicate a decreased CodeBLEU score.

One of the main advantages of using PEFT methods is that they require significantly less computing during training, mainly because of reduced backpropagation calculations. The fewer trainable parameters, the more efficient. Results in Figure 2 show a significant reduction of trainable parameters of PEFT methods compared to full fine-tuning. This is also essential to fit big models into GPU memory. For example, we could train the biggest 16 billion parameter model using a single NVIDIA A100 80GB GPU for LoRA, whereas we needed 6 x NVIDIA A100 40GB GPU for full fine-tuning.

Figure 3 shows the CodeBLEU score of each tuning method relative to the fraction of trainable parameters, symmetrically normalized between -1 and 1. As shown in Figure 3, prompt tuning has the highest efficiency among seven (i.e., CodeGen-350M-multi, CodeGen2-1B, StarCoder2-3B, CodeGen2-3.7B, StarCoder2-7B, StarCoder2-15B, and CodeGen2-16B) of the ten models and is the method that you get the ”most bang for your buck.” Moreover, prompt tuning has the most potential effect on models with large sizes, i.e., StarCoder2-15B and CodeGen2-16B. This can be explained by the increased capacity of larger models to understand better and leverage the additional context provided by prompt tuning. However, prompt tuning does not show consistent superiority for all the models. The method contributes to negative performance in three models (CodeLlama-7B, CodeGen2-7B, and StarCoder-Base).

For the three smallest models (CodeGen-350M-multi, CodeGen2-1B, and StarCoder2-3B), (IA)^3 clocks in at second place, which provides better results than LoRA. However, (IA)^3 fails to improve the three largest models. LoRA achieves a relatively low efficiency compared to the other PEFT methods. However, it does not lead to a negative impact on the performance of all models. Full fine-tuning obviously has the worst efficiency, as it needs to train all the model parameters.

Summary for RQ2: PEFT methods offer significantly reduced computational requirements compared to full fine-tuning and are ranked by resource utilization effectiveness as follows: prompt tuning $>$ (IA)^3 $>$ LoRA $>$ full fine-tuning. Prompt tuning demonstrates the highest returns on investment for most models but also carries risks of negative performance impact.

VI. Discussion

This section highlights our novel insights by comparing our results with related work. We also discuss possible threats to the validity of our study and how we mitigate them.

A. Comparison with related works

Unlike existing studies, we investigated using PEFT and LLM to generate complete unit tests because the effectiveness of PEFT methods is task-dependent. Our results provide novel insights into their performance in the context of unit test generation. For example, Ayupov and Chirkova [5] conclude that LoRA is less effective than full fine-tuning for code generation. We show contradictory results that LoRA is more effective than fine-tuning in unit test generation. Weyssow et al. [8] explored generating code from natural language instructions and concluded that LoRA is the most effective one in the PEFT methods they evaluate. Our results show that prompt tuning is the most effective in terms of cost and resource utilization in unit test generation, although LoRA is more effective without considering the cost aspects.

Compared to the existing studies, we also performed a more comprehensive evaluation, as we invested significant efforts to compare all PEFT methods to full fine-tuning across all model sizes (up to 16B parameters). In contrast, Ding et al. [24] only evaluated PEFT using the 355 million parameter RoBERTa$_{\text{LARGE}}$ model, while Weyssow et al. [8] only did full fine-tuning for smaller models up to maximum 350 million parameters.

Several studies have leveraged LLMs for generating unit tests [1, 2, 3, 4, 29]. However, none of the works investigate the potential performance of using PEFT compared to full fine-tuning. Tufano et al. [1] automatically generated unit tests and reported 95% syntactic correctness when training BART large (400 million parameters) [52] on METHODS2TEST. With PEFT methods, we can generate an average median of 94.03% syntactically correct unit tests. Training between 0.02 million and 13 million parameters on 1% of the same data showcasing the potential powerful performance of PEFT methods. Our results also illustrate that LoRA surpasses full fine-tuning in terms of boosting the quality of the generated unit tests measured using CodeBLU in several cases.

B. Implications to Academia and Industry

Writing unit tests is a hard and often time-consuming task [53]. However, they are essential to ensure code quality, stability, and maintainability [54]. Automating unit test generation using LLMs can significantly reduce the time and effort required for this process, making it an attractive solution for academia and industry. However, as our experiments show, full fine-tuning is sometimes required to achieve adequate quality of the generated unit tests, which is computationally demanding. Many LLM users may not have the necessary computing resources to fine-tune LLMs fully. Our results show that various PEFT methods can achieve similar performance as full fine-tuning by training a minuscule parameter fraction. Furthermore, our results suggest starting with prompt tuning for cost-effectiveness. Then, if stability issues arise, LoRA should be considered as a reliable alternative because it does not lead to negative performance impact.

C. Threaths to Validity

1) External validity: One of the threats to external validity is the selection of models. We minimize this threat by selecting a broad range of models varying in size, pre-training data, and learning objectives. Another potential concern is the validity of the METHODS2TEST$_{\text{SMALL}}$ benchmarking dataset. We opted for this dataset over other alternatives because it comprises real-world unit tests. We also validate the Java-translated version of the popularly used HumanEval code benchmarking dataset to mitigate the threat. The last threat is related to our evaluation metric. While popularly used, we recognize BLEU as a sub-optimal performance estimator for code generation. We, therefore, employ a more code-friendly version, CodeBLEU. Although CodeBLEU may not be the optimal metric to evaluate the functional correctness of unit tests completely, it provides a good tradeoff between evaluation effort and validity. To supplement our evaluation, we also calculate pass@1 and code coverage for the METHODS2TEST$_{\text{SMALL}}$ dataset, as this contains isolated code samples that are easier to test.

2) Internal validity: The main threat to internal validity is comprised of the selection of hyperparameters. We use the hyperparameters reported in the introductory paper of each PEFT method. Additionally, we use greedy decoding to support reproducibility and reduce variability in the generated results.

VII. Conclusion and future work

Although PEFT has been investigated to boost the quality and efficiency of code-related tasks using LLMs, they have not been studied in the context of unit test generation. We performed the first empirical study to compare different PEFT methods and full fine-tuning to generate unit tests. Our comparison involved ten LLMs of different sizes and architectures. Our results offer valuable insights into the advantages and challenges of using PEFT and full fine-tuning for unit test generation. With the knowledge we've gathered, individuals can choose the most suitable PEFT methods or full fine-tuning based on the size and characteristics of the LLMs.

In addition to unit tests, manually coding other automated test cases, such as integration and system tests, can be cost-intensive. In the future, we intend to explore the utilization of LLMs in conjunction with cost-effective tuning methods to automatically generate other types of tests.

VIII. Data availability

The METHODS2TEST$_{\text{SMALL}}$ dataset is available at:
https://huggingface.co/datasets/andstor/methods2test_small

All fine-tuned models are also made available at:
https://huggingface.co/andstor/peft-unit-test-generation-experiments

Experiment code and results are available at:
https://github.com/andstor/peft-unit-test-generation-replication-package

References

[1] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," 2021.

[2] M. L. Siddiq, J. C. S. Santos, R. H. Tanvir, N. Ulfat, F. A. Rifat, and V. C. Lopes, "Using large language models to generate junit tests: An empirical study," 2024.

[3] Z. Yuan *et al.*, "No more manual tests? evaluating and improving chatgpt for unit test generation," 2023.

[4] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," 2023.

[5] S. Ayupov and N. Chirkova, "Parameter-efficient fine-tuning of transformers for source code," *ArXiv*, vol. abs/2212.05901, 2022.

[6] D. Goel, R. Grover, and F. 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*, ser. ICPC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 71–81. [Online]. Available: https://doi.org/10.1145/3524610.3527892

[7] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "Prompt tuning in code intelligence: An experimental evaluation," *IEEE Transactions on Software Engineering*, vol. 49, pp. 4869–4885, 2023.

[8] M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui, "Exploring parameter-efficient fine-tuning techniques for code generation with large language models," 2024.

[9] D. Wang *et al.*, "One adapter for all programming languages? adapter tuning for code search and summarization," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 5–16.

[10] E. Shi *et al.*, "Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 39–51. [Online]. Available: https://doi.org/10.1145/3597926.3598036

[11] B. Liu *et al.*, "Mftcoder: Boosting code llms with multitask fine-tuning," *ArXiv*, vol. abs/2311.02303, 2023.

[12] Z. Yuan, J. Liu, Q. Zi, M. Liu, X. Peng, and Y. Lou, "Evaluating instruction-tuned large language models on code comprehension and generation," 2023.

[13] J. Liu, C. Sha, and X. Peng, "An empirical study of parameter-efficient fine-tuning methods for pre-trained code models," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2023, pp. 397–408. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ASE56229.2023.00125

[14] A. Mastropaolo *et al.*, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 336–347.

[15] Y. Yu *et al.*, "Low-rank adaptation of large language model rescoring for parameter-efficient speech recognition," in *2023 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, 2023, pp. 1–8.

[16] H. Liu *et al.*, "Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning," *ArXiv*, vol. abs/2205.05638, 2022.

[17] B. Lester, R. Al-Rfou, and N. Constant, "The power of scale for parameter-efficient prompt tuning," in