# No More Fine-Tuning? An Experimental Evaluation of Prompt Tuning in Code Intelligence

Chaozheng Wang
Harbin Institute of Technology
Shenzhen, China
wangchaozheng@stu.hit.edu.cn

Yuanhang Yang
Harbin Institute of Technology
Shenzhen, China
ysngkil@gmail.com

Cuiyun Gao*
Harbin Institute of Technology
Shenzhen, China
gaocuiyun@hit.edu.cn

Yun Peng
The Chinese University of Hong Kong
Hong Kong, China
ypeng@cse.cuhk.edu.hk

Hongyu Zhang
The University of Newcastle
Newcastle, Australia
hongyu.zhang@newcastle.edu.au

Michael R. Lyu
The Chinese University of Hong Kong
Hong Kong, China
lyu@cse.cuhk.edu.hk

## ABSTRACT

Pre-trained models have been shown effective in many code intelligence tasks. These models are pre-trained on large-scale unlabeled corpus and then fine-tuned in downstream tasks. However, as the inputs to pre-training and downstream tasks are in different forms, it is hard to fully explore the knowledge of pre-trained models. Besides, the performance of fine-tuning strongly relies on the amount of downstream data, while in practice, the scenarios with scarce data are common. Recent studies in the natural language processing (NLP) field show that prompt tuning, a new paradigm for tuning, alleviates the above issues and achieves promising results in various NLP tasks. In prompt tuning, the prompts inserted during tuning provide task-specific knowledge, which is especially beneficial for tasks with relatively scarce data. In this paper, we empirically evaluate the usage and effect of prompt tuning in code intelligence tasks. We conduct prompt tuning on popular pre-trained models CodeBERT and CodeT5 and experiment with three code intelligence tasks including defect prediction, code summarization, and code translation. Our experimental results show that prompt tuning consistently outperforms fine-tuning in all three tasks. In addition, prompt tuning shows great potential in low-resource scenarios, e.g., improving the BLEU scores of fine-tuning by more than 26% on average for code summarization. Our results suggest that instead of fine-tuning, we could adapt prompt tuning for code intelligence tasks to achieve better performance, especially when lacking task-specific data.

## CCS CONCEPTS

• **Software and its engineering → Software development techniques**;

## KEYWORDS

code intelligence, prompt tuning, empirical study

**ACM Reference Format:**
Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu. 2022. 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 (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549113

*Corresponding author. The author is also affiliated with Peng Cheng Laboratory and Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies.


*ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore*

ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549113

## 1 INTRODUCTION

Code intelligence leverages machine learning, especially deep learning (DL) techniques to mine knowledge from large-scale code corpus and build intelligent models for improving the productivity of computer programming. The state-of-the-art DL-based approaches to code intelligence exploit the *pre-training and finetuning* paradigm [1, 9, 14, 21, 56], in which language models are first pre-trained on a large unlabeled text corpora and then finetuned on downstream tasks. For instance, Feng *et al.* [9] propose CodeBERT, a pre-trained language model for source code, which leverages both texts and code in the pre-training process. To facilitate generation tasks for source code, Wang *et al.* [56] propose a pre-trained sequence-to-sequence model named CodeT5. These pre-trained source code models achieve significant improvement over previous approaches.

However, there exist gaps between the pre-training and fine-tuning process of these pre-trained models. As shown in Figure 1(a), pre-training models such as CodeBERT [9] and CodeT5 [56] are generally pre-trained using the Masked Language Modeling (MLM) objective. The input to MLM is a mixture of code snippets and natural language texts, and the models are trained to predict randomly-masked input tokens. However, when models are fine-tuned into the downstream tasks, e.g. defect detection, the input involves only source code and the training objective changes to a classification problem. As shown in Figure 1(b), the pre-trained model represents each input code snippet using a classification head (CLS Head) and fine-tunes the CLS head based on a task-specific dataset. The inconsistent inputs and objectives between pre-training and fine-tuning render the knowledge of pre-trained models hard to be fully explored, leading to sub-optimal results for downstream tasks. Besides, the performance of fine-tuning largely depends on the scale of downstream data [13, 16, 25, 59].

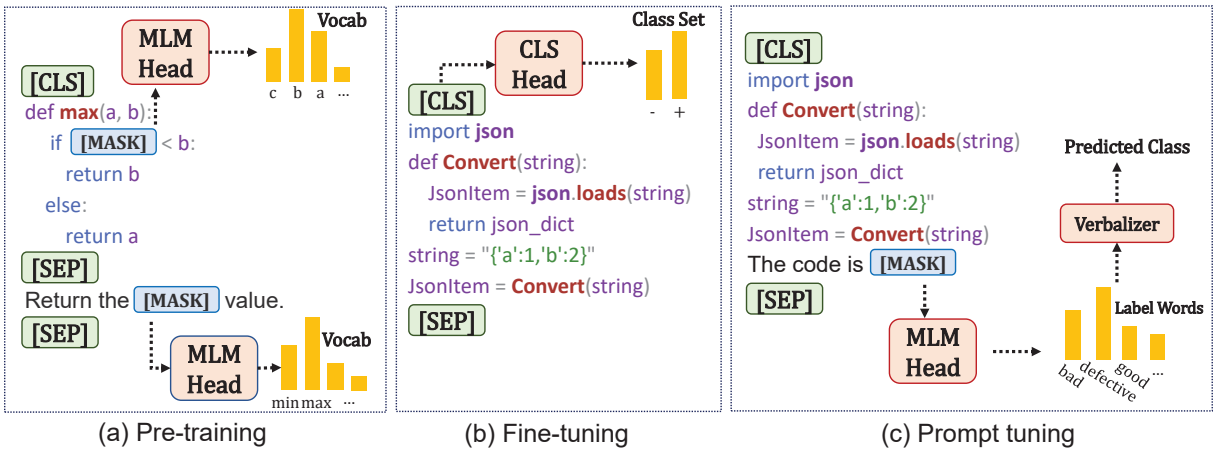


**Figure 1: Illustration on the process of pre-training, fine-tuning, and prompt tuning on defect detection task. [CLS] and [SEP] denote two special tokens in pre-trained models.**

Recently, prompt tuning [16, 25, 26, 33, 49] is proposed to mitigate the above issues of fine-tuning. Figure 1(c) illustrates the concept of prompt tuning. Instead of only involving source code as input, prompt tuning firstly rewrites the input by adding a natural language prompt such as "*The code is* [*MASK*]" at the end of the code snippet, and then let the model predict the masked token [*MASK*]. There is also a verbalizer [16, 49] that maps the tokens predicted by the model to the class. By adding a prompt and verbalizer, prompt tuning reformulates the classification problem into an MLM problem, aligning the objective with the pre-training stage. This alignment unleashes the hidden power stored in the pre-trained models. Besides, the inserted natural language prompt can involve task-specific knowledge to facilitate the adaption to downstream tasks [15, 26, 49, 50].

Inspired by the success of prompt tuning in the NLP field, we would like to investigate if prompt tuning is effective for code intelligence tasks, which to our best knowledge still remains unexplored. In this paper, we conduct an experimental evaluation on the effectiveness of prompt tuning on three popular code intelligence tasks: defect detection, code translation, and code summarization. We mainly investigate the following three research questions (RQs):

- How effective is the prompt tuning in solving code intelligence tasks?
- How capable is prompt tuning to handle data scarcity scenarios?
- How different prompt templates affect the performance of prompt tuning?

To answer the first RQ, we apply prompt tuning to the three code intelligence tasks. To answer the second RQ, we evaluate prompt tuning in data scarcity scenarios from two aspects, including low-resource settings and cross-domain settings. To answer the third RQ, we comprehensively study the influence of different prompt templates and verbalizers on model performance.

Based on the experiment results, we find that prompt tuning brings non-trivial improvement to the the performance of downstream code intelligence tasks, including both classification and generation tasks. Furthermore, prompt tuning can significantly outperform conventional fine-tuning, especially when the training data are scarce.

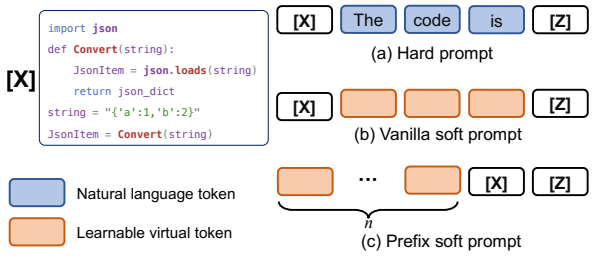


**Figure 2: Illustration on the different types of prompt, where [X] and [Z] indicate the input slot and answer slot, respectively. Both vanilla soft prompt (b) and prefix soft prompt (c) belong to soft prompt.**

The major contributions of this paper are as follows:

(1) To the best of our knowledge, this paper serves as the first study on the performance of prompt tuning for code intelligence tasks.
(2) We explore how different prompts can affect the performance of prompt tuning on code intelligence tasks.
(3) We discuss the implications of our findings and suggest further research on the usage of prompt tuning.

## 2 BACKGROUND

### 2.1 Fine-Tuning

Fine-tuning a pre-trained model for downstream tasks [6, 29, 40] is a prevalent paradigm in the NLP field. Fine-tuning aims at exploiting the knowledge learned by pre-trained models without learning from scratch and can be regarded as a way of applying transfer learning [39]. To adapt pre-trained models into downstream tasks, fine-tuning trains the model in a supervised way. Specifically, given

a dataset which consists of task-specific samples $X$ and corresponding labels $Y$, fine-tuning aims to find a set of parameters $\theta$ for the pre-trained model, that $\theta = \arg\min_{\theta} P(Y|X;\theta)$.

## 2.2 Prompt Tuning

The intuition of prompt tuning is to convert the training objective of downstream tasks into a similar form as the pre-training stage, i.e., the MLM objective [6, 9, 34]. As shown in Figure 1(c), prompt tuning aims at predicting masked tokens in the input. It also modifies the model input by adding a natural language prompt, enabling the input format identical to the pre-training stage.

Specifically, prompt tuning employs a prompt template $f_{prompt}(x)$ to reconstruct the original input $\boldsymbol{x}$, producing new input $\boldsymbol{x'}$. As illustrated in Figure 5, the prompt template can involve two types of reserved slots in, i.e., input slot $[X]$ and answer slot $[Z]$. The input slot $[X]$ is reserved to be filled with original input text, and the answer slot $[Z]$ is to be filled by predicted labels such as *defective*. For the example shown in Figure 1, prompt tuning outputs the final predicted class by a verbalizer [16, 49]. The verbalizer, denoted as $\mathcal{V}$, is an injective function which maps each predicted label word to a class in the target class set $Y$:

$$\mathcal{V} : W \rightarrow Y \tag{1}$$

where $W$ indicates the label word set. For the example in Figure 1 (c), the label word set $W$ includes "*[bad, defective]*" for buggy code snippets and "*[perfect, clean]*" for the others. The class set $Y$ contains "+" and "−" for indicating defective and clean code, respectively. In the example, the verbalizer maps the label with highest probability "*defective*" into the target class "+" in the class set.

According to the flexibility of the inserted prompt, prompt tuning techniques can be categorized into two types: hard prompt and soft prompt. We elaborated on the details of each prompt type in the following.

*2.2.1 Hard Prompt.* The *hard prompt* [13, 16, 49] is a technique that modifies the model input by adding fixed natural language instruction (prompts). It aims to elicit task-specific knowledge learned during pre-training for the tuning stage. Hard prompt is also known as *discrete prompt* since each token in the prompts is meaningful and understandable [13, 30]. For instance, in the defect detection task, by appending "The code is [Z]." to the input code, the task objective becomes predicting the label word at the answer slot $[Z]$, such as "*defective*" or "*clean*". The designed prompt template for defect prediction task can be formulated as:

$$f_{prompt}(x) = \text{"}[X]\ \textit{The code is}\ [Z]\text{"} \tag{2}$$

where $[X]$ denotes the input code. Although hard prompt has shown promising performance in previous work, the template design and the verbalizer choices are challenging. For example, the prompt template $f_{prompt}(x)$ can also be designed as "$[X]$ *It is* $[Z]$", where the label words in the verbalizer involve "*bad*" and "*perfect*".

*2.2.2 Soft Prompt.* The *Soft prompt* [16, 26, 53], as the name implies, is an alternative to hard prompt. Different from hard prompt, the tokens in the soft prompt template are not fixed discrete words of a natural language. Instead, these tokens are continuous vectors which can be learnt during the tuning stage. They are also called *virtual tokens* because they are not human-interpretable. Soft prompt is proposed to alleviate the burden of manually selecting prompt template in hard prompt. There are two kinds of soft prompt, denoted as *vanilla soft prompt* and *prefix soft prompt*, respectively.

*Vanilla soft prompt*, as depicted in Figure 2(b), can be obtained by simply replacing the hard prompt token with a virtual one, denoted as $[SOFT]$, such as:

$$f_{prompt}(x) = \text{"}[X]\ [SOFT]\ [SOFT]\ [SOFT]\ [Z]\text{"} \tag{3}$$

The embedding of virtual tokens are optimized during tuning stage.

*Prefix soft prompt* prepends several virtual tokens to the original input, as shown in Figure 2(c). It can generate comparable performance with the vanilla soft prompt and hard prompt.

$$f_{prompt}(x) = \text{"}[SOFT] * n\ [X]\ [Z]\text{"} \tag{4}$$

where $n$ indicates the number of virtual tokens.

# 3 EXPERIMENTAL EVALUATION

## 3.1 Research Questions

We aim at answering the following research questions through an extensive experimental evaluation:

- **RQ1:** How effective is the prompt tuning in solving code intelligence tasks?
- **RQ2:** How capable is prompt tuning to handle data scarcity scenarios?
- **RQ3:** How different prompt templates affect the performance of prompt tuning?

We design RQ1 to verify our hypothesis that prompt tuning, which aligns the training objectives with the pre-training stage, is more effective than fine-tuning for the downstream code intelligence tasks. RQ2 aims at investigating whether prompt tuning embodies advantage in data scarcity scenarios including low-resource and cross-domain settings. In RQ3, we aim at exploring the impact of different prompt templates, such as varying prompt types and selection of label words, on the performance of downstream tasks.

**Table 1: Statistics of the datasets used in this paper.**

| Tasks | Datasets | Training Set | Val. Set | Test Set |
|---|---|---|---|---|
| Defect Detection | Defect | 21,854 | 2,732 | 2,732 |
| Code Summarization | Ruby | 48,791 | 2,209 | 2,279 |
| | JavaScript | 123,889 | 8,253 | 6,483 |
| | Go | 317,832 | 14,242 | 14,291 |
| | Python | 409,230 | 22,906 | 22,104 |
| | Java | 454,451 | 15,053 | 26,717 |
| | PHP | 523,712 | 26,015 | 28,391 |
| Code Translation | Translation | 10,300 | 500 | 1,000 |

## 3.2 Code Intelligence Tasks with Prompt Tuning

To evaluate the prompt tuning technique on source code, we adopt three downstream code intelligence tasks, namely defect detection,

code summarization, and code translation. We describe the detail of pre-trained models and prompt template of each task in the following.

*3.2.1 Pre-trained Models.* We choose CodeBERT [9] and CodeT5 [56] as the studied pre-trained models, since they are the most widely-used model and state-of-the-art model for source code, respectively.

**CodeBERT** [9] is an encoder-only model which is realized based on RoBERTa [34]. CodeBERT is pre-trained on CodeSearchNet [18]. It is able to encode both source code and natural language text. CodeBERT has 125 million parameters.

**CodeT5** [56], a variant of text to text transfer Transformer [45], is the state-of-the-art model for code intelligence tasks. It regards all the tasks as sequence to sequence paradigm with different task specific prefixes. It can solve both code understanding and code generation tasks. Code-T5 is pre-trained on a larger dataset including CodeSearchNet [18] and an additional C/C# language corpus collected by the authors. CodeT5 is classified into two versions: CodeT5-small and CodeT5-base, according to their sizes. The numbers of parameters in CodeT5-small and CodeT5-base are 60 million and 220 million, respectively.

*3.2.2 Defect Detection.* Given a code snippet, defect detection [28, 62] aims to identify whether it is defect prone, such as memory leakage and DoS attack. The task is defined as a binary classification task in training CodeBERT and a generation task in training CodeT5 [45, 56].

For *hard prompt*: As shown in Figure 1(c), with prompt tuning, models predict the probability distribution over the label words. A verbalizer $\mathcal{V}$ maps the label word with highest probability to the predicted class. One cloze-style template $f_{prompt}(\cdot)$ with an input slot $[X]$ and an answer slot $[Z]$ is designed as below:

$$f_{prompt}(x) = \text{“}The\ code\ [X]\ is\ [Z]\text{”}$$
$$\mathcal{V} = \begin{cases} + : & [defective, bad] \\ - : & [clean, perfect] \end{cases} \tag{5}$$

where the left and right sides of : indicate the predicted class and corresponding label words. To study the impact of different prompts, we also design other prompt templates including "$[X]$ *It is* $[Z]$", "$[X]$ *The code is* $[Z]$", "$[X]$ *The code is defective* $[Z]$" and "*A* $[Z]$ *code* $[X]$".

For *vanilla soft prompt*: For facilitating the comparison of hard prompt and vanilla soft prompt, we simply replace the natural language tokens in the hard prompt templates with virtual tokens for generating vanilla soft prompts. For example, "$[X][SOFT][SOFT][Z]$" is the vanilla soft prompt version of "$[X]$ *It is* $[Z]$".

For *prefix soft prompt*: We design the prefix soft prompt by appending a learnable prefix prompt according to Equation (4).

*3.2.3 Code Summarization.* Given a code snippet, the code summarization task aims to generate a natural language comment to summarize the functionality of the code. We only utilize CodeT5 in this task because CodeBERT does not have a decoder to generate comments.

For *hard prompt*: We append the natural language instruction of the task to the input code, so the template can be:

$$f_{prompt}(x) = \text{“}\ Generate\ comment\ for\ [LANG]\ [X]\ [Z]\text{”} \tag{6}$$

where $[LANG]$, $[X]$, and $[Z]$ denote the slot of programming language type, input slot, and the generated answer slot. The natural language instruction "*Generate comment for*" is manually predefined for adjusting the generation behavior of CodeT5. We also design other prompt templates for experimentation including *Summarize [LANG]*. Note that there is *no verbalizer* for the generation task.

For the *vanilla soft prompt* and *prefix soft prompt*: They are designed in the same way as the defect detection task. For example, we replace the natural language tokens in the hard prompt templates into virtual tokens for generating vanilla soft prompts. The prefix soft prompts are defined according to Equation (4).

*3.2.4 Code Translation.* Code translation aims to migrate legacy software from one programming language to another one. The *vanilla soft prompt* and *prefix soft prompt* are designed similar to the above two tasks, so we only describe about the *hard prompt* for the task. For *hard prompt*, we design the template by appending task-specific instruction:

$$f_{prompt}(x) = \text{“}Translate\ [X]\ to\ [LANG]\ [Z]\text{”} \tag{7}$$

The template explains that the model is translating the input code $[X]$ in one programming language to the code $[Z]$ in another programming language $[LANG]$.

## 3.3 Evaluation Datasets

To empirically evaluate the performance of prompt tuning for source code, we choose the datasets for the three tasks from the popular CodeXGLUE benchmark[1] [36].

*3.3.1 Defect Detection.* The dataset is provided by Zhou *et al.* [62]. It contains 27K+ C code snippets from two open-source projects QEMU and FFmpeg, and 45.0% of the entries are defective.

*3.3.2 Code Summarization.* We use the same dataset as the CodeT5 work [56]. The dataset is from CodeSearchNet [18], which contains thousands of code snippet and natural language description pairs for six programming languages including Python, Java, JavaScript, Ruby, Go and PHP.

*3.3.3 Code Translation.* The dataset is provided by Lu *et al.* [36], and is collected from four public repositories (including Lucene, POI, JGit and Antlr). Given a piece of Java (C#) code, the task is to translate the code into the corresponding C# (Java) version.

## 3.4 Evaluation Metrics

*3.4.1 Defect Detection:* For the defect detection task, following [56], we use *Accuracy* as the evaluation metric. The metric is to measure the ability of model to identify insecure source code, defined as:

$$ACC = \frac{\sum_{i=1}^{|D|} 1(y_i == \hat{y}_i)}{|D|} \tag{8}$$

[1] https://github.com/microsoft/CodeXGLUE

**Table 2: Hyperparameter settings**

| Hyperparameter | Value | Hyperparameter | Value |
|---|---|---|---|
| Optimizer | AdamW[35] | Warm up steps | 10% |
| Learning rate | 5e-5 | Training batch size | 64 |
| LR scheduler | Linear | Validation batch size | 64 |
| Beam size | 10 | Adam epsilon | 1e-8 |
| Max. gradient norm | 1.0 | | |

where $D$ is the dataset and $|D|$ denotes its size. The symbol $y_i$ and $\hat{y}_i$ indicate the ground truth label and predicted label, respectively. The $1(x)$ function returns 1 if $x$ is True and otherwise returns 0.

*3.4.2 Code Summarization:* Following previous work [9, 56], we use Bilingual Evaluation Understudy (BLEU) score to evaluate the quality of generated comments. The idea of BLEU is that the closer the generated text is to the result of ground truth text, the higher the generation quality. The metric is defined as below:

$$BP = \begin{cases} 1 & if\ c > r \\ e^{1-r/c} & if c \leq r \end{cases} \tag{9}$$

$$BLEU = BP \cdot exp\Big(\sum_{n=1}^{N} w_n \log p_n\Big) \tag{10}$$

where $p_n$ means the modified n-gram precision and $w_n$ is the weight. $BP$ represents the brevity penalty, and $c$ and $r$ indicate the lengths of generated comment and target comment length, respectively. In our experiments, we choose smoothed BLEU-4 score, i.e., $n = 4$, for evaluating the generation tasks following previous work [9, 56].

*3.4.3 Code Translation:* To better measure the quality of generated code snippets, besides BLEU score, another two metrics including Accuracy and CodeBLEU [46] are used following [36, 56]. The computation of Accuracy is the same as Equ. (8), which is the most strict metric.

CodeBLEU parses the generated code, and takes both the code structure and semantics into account for measuring the similarity between the generated code and the code in ground truth. Its computation consists of four components including n-gram matching score ($BLEU$), weighted n-gram matching score $weighted\_BLEU$, syntactic AST matching score $AST\_Score$ and semantic data flow matching score $DF\_Score$:

$$\begin{aligned} CodeBLEU = \alpha * BLEU + \beta * weighted\_BLEU \\ + \gamma * AST\_Score + \delta * DF\_Score \end{aligned} \tag{11}$$

where $\alpha, \beta, \gamma, \delta$ are weights for each component. Following [36, 56], they are all set as 0.25.

## 3.5 Implementation Details

*3.5.1 Experimental Setup.* All the pre-trained models and corresponding tokenizer in our experimentation are loaded from the official repository Huggingface[2]. The overall framework is Pytorch[3]. Our implementation of prompt is based on OpenPrompt [7]. We use the generic training strategy and parameter settings following the official implementation of CodeT5 [56], with details shown in Table 2.

Specifically, for the defect detection task, we train CodeBERT and CodeT5 for 5 and 15 epochs, respectively. For CodeT5 model, we set the maximum source length and target length as 512 and 3, respectively. For the code summarization task, because CodeBERT is an encoder-only architecture model, we focus on evaluating prompt tuning on CodeT5. We train CodeT5 for 20 epochs. The maximum lengths of source text and target text are defined as 256 and 128. For the code translation tasks, we train the CodeT5 models for 50 epochs. The maximum length of source text and target text are set as 256 and 256, respectively.

For parameter configuration in fine-tuning, we use the configurations provided by the original work [9, 56], which were already well adjusted. For a fair comparison, we use the same parameter configurations when implementing prompt tuning.

All the experiments are run on a server with 4 * Nvidia Tesla V100 and each one has 32GB graphic memory.

*3.5.2 Fine-Tuning Baselines.* We fine-tuned CodeBERT and CodeT5 on the three code intelligence tasks. Specifically, we fine-tune CodeBERT only for the defect detection and CodeT5 for all the three tasks. For CodeBERT, we use the first output token (the [CLS] token) as the sentence embedding and feed it into a feed-forward network (FFN) to generate predictions. For CodeT5, all the tasks are treated as generation tasks. It takes either code or natural language sentences as input and generate target texts.

**Table 3: Classification accuracy on defect detection.**

| Methods | | Accuracy |
|---|---|---|
| CodeBERT | Fine-tuning | 62.12 |
| | Prompt tuning | **64.17** |
| CodeT5-small | Fine-tuning | 62.96 |
| | Prompt tuning | **63.91** |
| CodeT5-base | Fine-tuning | 65.00 |
| | Prompt tuning | **65.82** |

# 4 EXPERIMENTAL RESULTS

## 4.1 RQ1: Effectiveness of Prompt Tuning

In this section, we study the effectiveness of prompt tuning by comparing with the standard tuning paradigm – fine-tuning on the three code intelligence tasks: defect detection, code summarization, and code translation. We present the best performance achieved by our experimented prompts. Full results can be found in our project repository[4]. We also discuss the impact of different prompts in Section 4.3.

**Defect Detection.** Table 3 shows the comparison results for defect detection, in which CodeBERT and CodeT5 serve as pre-trained models. We can observe that prompt tuning always outperforms fine-tuning across different pre-trained models. For example, prompt tuning obtains an improvement of 3.30% over fine-tuning on CodeBERT. For CodeT5-small and CodeT5-base, the improvements

[2]https://huggingface.co/models
[3]https://pytorch.org/
[4]https://github.com/adf1178/PT4Code

**Table 4: Results (BLEU-4 scores) of the CodeT5 model on code summarization task.**

| Methods | | Ruby | JavaScript | Go | Python | Java | PHP | Overall |
|---|---|---|---|---|---|---|---|---|
| CodeT5-small | Fine-tuning | 13.38 | 14.94 | 21.27 | 17.88 | 18.38 | 24.70 | 18.43 |
| | Prompt tuning | **13.60** | **15.91** | **22.33** | **18.34** | **20.60** | **26.95** | **19.62** |
| CodeT5-base | Fine-tuning | 13.70 | 15.80 | 22.60 | 17.97 | 19.56 | 25.77 | 19.23 |
| | Prompt tuning | **14.29** | **16.04** | **23.11** | **18.52** | **19.72** | **27.06** | **19.79** |

```
public virtual int size(){
  lock (mutex){
    return c.size();
  }
}
```
(a) Original C# code

```
@Override public int size() {
  synchronized (mutex) {
    return c.size();
  }
}
```
(b) Ground truth Java code

```
public int size() {
  return c.size();
}
```
(c) Generated Java code by fine-tuning

```
public int size() {
  synchronized (mutex) {
    return c.size();
  }
}
```
(d) Generated Java code by prompt tuning

**Figure 3: An example for illustrating the quality of code snippets translated by fine-tuning and prompt tuning in the code translation task, respectively, where the pre-trained model is CodeT5-small.**

are 1.51% and 1.26%, respectively. We also perform a statistical significance test (t-test) on defect detection task, and the results show that prompt tuning outperforms fine-tuning at the significance level at 0.05 (p-value 0.048). The results indicate that prompt tuning is more effective than fine-tuning for pre-trained models with different architecture or different sizes on the defect detection task.

**Code Summarization.** Since CodeBERT is an encoder-only model, we only involve CodeT5 as the pre-trained model on the code summarization task. Table 4 presents the BLEU-4 scores achieved by prompt tuning and fine-tuning for different programming languages. We can observe consistent improvement on overall performance as in the defect detection task: compared with fine-tuning, prompt tuning obtains an improvement of 6.46% and 2.91% when using CodeT5-small and CodeT5-base as pre-trained models, respectively. Looking into specific programming language, prompt tuning also always achieves better summarization performance than fine-tuning. It shows the largest advancement for the code written in PHP, with increase rate at 9.11% and 5.01% on CodeT5-small and CodeT5-base, respectively. Moreover, prompt tuning can perform statistically better than fine-tuning at the significance level 0.05 on code summarization with a p-value 0.019. The results indicate the effectiveness of prompt tuning in the code summarization task.

**Code Translation.** For the task, we only involve the pre-trained CodeT5 model for evaluating the performance of prompt tuning. The results of prompt tuning and fine-tuning based on CodeT5 are depicted in Table 5. From the table, we can see that prompt tuning outperforms fine-tuning in both directions of translation. Comparing with fine-tuning, prompt tuning achieves an average improvement of 1.22%, 0.85%, and 0.87% on both directions for BLEU, Accuracy and CodeBLEU, respectively. The improvement demonstrates the effectiveness of prompt tuning on this task. To better illustrate how prompt tuning improves the quality of code translation, we give an example in Figure 3. From the example, we can see that using fine-tuning methods, CodeT5-small does not accurately translate the C# code into the corresponding Java version by missing an important synchronized lock statement *"synchronized (mutex)"*, while it can output more accurately with prompt tuning. We attribute this improvement to the learned prior knowledge carried by the prefix soft prompts. Through the powerful prior knowledge, CodeT5 can quickly adapt to the translation of the code in C# to Java, and pay more attention to language-specific grammar details. But fine-tuning methods can only make the model learn the translation direction after multiple iterations of training, the model may fail to focus on the important part such as "*lock*" in the input.

Based on the performance of all the three tasks, we find that prompt tuning is more effective than fine-tuning. Another interesting observation is that the improvement of prompt tuning on CodeT5-small is 1.51%, 6.46%, and 1.22%, respectively, which is higher than that on CodeT5-base, with the increase rat at 1.26%, 2.91%, and 0.43%, respectively. This may be attributed to that CodeT5-base is a significantly larger model than CodeT5-small (220 million v.s. 60 million parameters), and prompt tuning (768 parameters per token). The observation suggests that prompt tuning shows more obvious improvement than fine-tuning for smaller pre-trained models.

**Finding 1:** Prompt tuning is more effective than fine-tuning on the code intelligence tasks, with respect to different pre-trained models and different programming languages. Besides, the advantage of prompt tuning is more obvious for smaller pre-trained models.

## 4.2 RQ2: Capability of Prompt Tuning in Different Data Scarcity Scenarios

Considering that the performance of fine-tuning is known to strongly rely on the amount of downstream data [10, 15, 38], while scenarios with scarce data in source code are common [4, 47, 51]. In this section, we study how well prompt tuning can handle the data scarcity scenarios. We focus on two kinds of data scarcity settings: 1) low-resource scenario, in which there are significantly few training instances, and 2) cross-domain scenario, in which the model is trained on a similar data-sufficient domain and tested on target domain.

**Performance in low-resource scenario.** We study the performance of prompt tuning in low-resource setting on the classification task, i.e., defect detection, and one generation task, i.e., code summarization. We simulate this setting by randomly select a small subset of training instances (also called *shots*) in the original dataset.

**Table 5: Experimental results on code translation tasks: Java-C# and C#-Java.**

| Methods | | C# to Java | | | Java to C# | | |
|---|---|---|---|---|---|---|---|
| | | BLEU | Accuracy | CodeBLEU | BLEU | Accuracy | CodeBLEU |
| CodeT5-small | Fine-tuning | 78.67 | 65.40 | 82.55 | 82.29 | 63.80 | 87.01 |
| | Prompt tuning | **79.59** | **66.00** | **83.06** | **83.33** | **64.30** | **87.99** |
| CodeT5-base | Fine-tuning | 79.45 | **66.10** | 83.96 | 83.61 | 65.30 | 88.32 |
| | Prompt tuning | **79.76** | **66.10** | **84.39** | **83.99** | **65.40** | **88.74** |

**Table 6: Classification accuracy (%) on defect detection in low-resource scenario. '-' denotes the model fails to converge due to extreme lack of training data.**

| Method | | Zero shot | 16 shots | 32 shots | 64 shots | 128 shots |
|---|---|---|---|---|---|---|
| CodeBERT | Fine-tuning | 50.52 | 52.15 | 53.01 | 53.61 | 55.28 |
| | Prompt tuning | **53.99** | **52.98** | **53.83** | **54.28** | **56.19** |
| CodeT5-small | Fine-tuning | - | - | 51.22 | 52.10 | 54.28 |
| | Prompt tuning | - | - | **52.36** | **53.59** | **55.04** |
| CodeT5-base | Fine-tuning | - | - | 51.25 | 52.64 | 54.52 |
| | Prompt tuning | - | - | **52.44** | **53.82** | **55.47** |

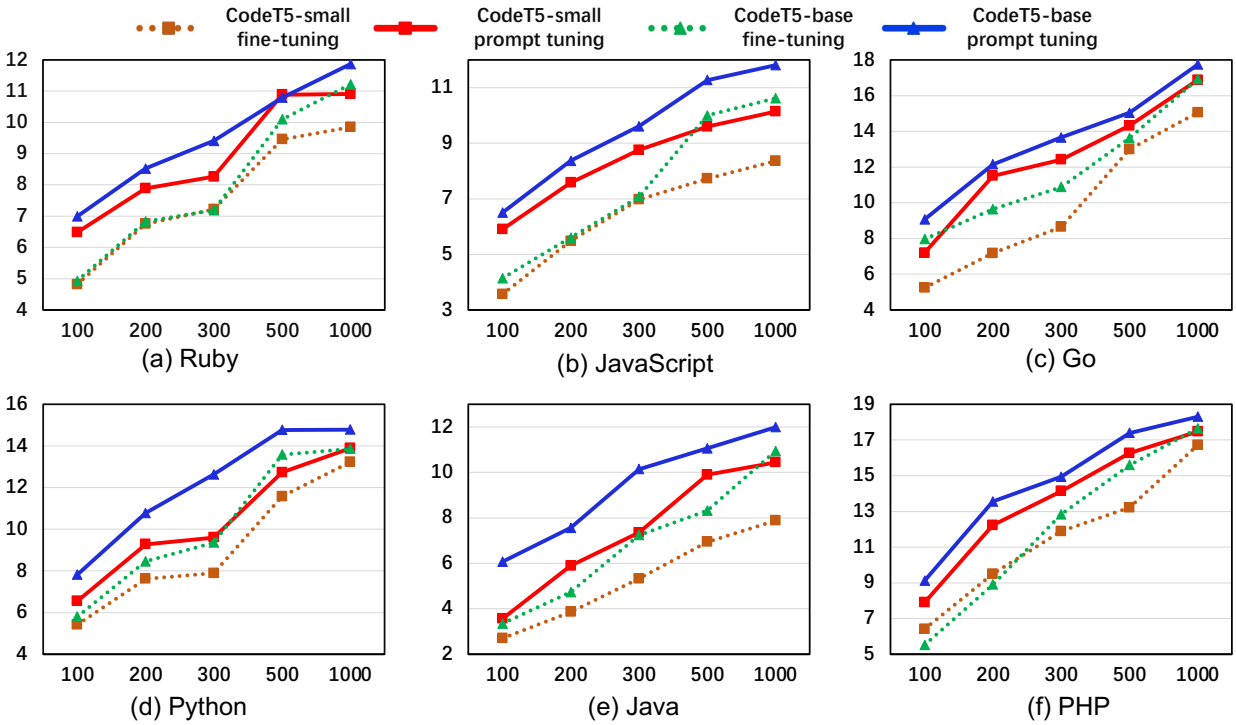


**Figure 4: Results of fine-tuning and prompt tuning on code summarization task in low resource scenarios. The horizontal axis indicates the number of training instances while the vertical axis means the BLEU-4 score.**

To avoid randomness in data selection, we produce each subset five times with different seeds and run four times on each dataset. The average results are reported.

For the defect detection task, we choose 0, 16, 32, 64, and 128 training shots per class to create five small training subsets. Table 6 presents the accuracy achieved by prompt tuning and fine-tuning regarding the five settings. Note that in zero-shot settings, no tuning data are involved. Given test data, the fine-tuning model directly generates target labels (defective or clean); while the prompt tuning model predicts the label words. By comparing the results with

**Table 7: Experimental results (BLEU-4 score) on cross-language code summarization. The models are trained on Python or Java datasets, and tested on Ruby, JavaScript and Go, respectively.**

| Training | Methods | Ruby | JavaScript | Go |
|---|---|---|---|---|
| CodeT5-small | | | | |
| Python | Fine-tuning | 12.75 | **12.37** | 11.57 |
| | Prompt tuning | **13.01** | 12.35 | **12.15** |
| Java | Fine-tuning | 12.20 | 11.45 | 10.96 |
| | Prompt tuning | **12.59** | **11.84** | **11.15** |
| CodeT5-base | | | | |
| Python | Fine-tuning | 13.06 | 12.81 | 12.89 |
| | Prompt tuning | **13.37** | **13.11** | **14.27** |
| Java | Fine-tuning | 12.67 | 11.50 | 11.88 |
| | Prompt tuning | **13.13** | **11.99** | **12.96** |

**Table 8: Classification accuracy (%) of comparing the performance of CodeBERT model on defect detection task via different prompt templates. The verbalizer is fixed as +: "*bad*", "*defective*"; −:"*perfect*", "*clean*". The underlined texts are replaced by virtual tokens in the corresponding vanilla soft prompt.**

| Hard Prompt | Vanilla Soft Prompt | Accuracy | |
|---|---|---|---|
| | | Hard | Soft |
| $[X]$ The code is $[Z]$ | $[X]$ $[SOFT] * 3$ $[Z]$ | 63.68 | 63.15 |
| A $[Z]$ code $[X]$ | $[SOFT]$ $[X]$ $[SOFT]$ $[Z]$ | 63.36 | 62.95 |
| $[X]$ It is $[Z]$ | $[X]$ $[SOFT][SOFT]$ $[Z]$ | 63.92 | 63.39 |
| The code $[X]$ is $[Z]$ | $[SOFT] * 2$ $[X]$ $[SOFT]$ $[Z]$ | **64.17** | 63.34 |

**Table 9: Classification accuracy (%) of different verbalizers on the defect detect task, where the pre-trained model is CodeBERT. The template is "The code $[X]$ is $[Z]$".**

| Verbalizer | Accuracy |
|---|---|
| +: "Yes" −: "No" | 63.08 |
| +: "bad" −: "perfect" | 63.71 |
| +: "bad", "defective" −: "clean", "perfect" | 64.17 |
| +: "bad", "defective", "insecure"<br>−: "clean", "perfect", "secure" | 63.26 |
| +: "bad", "defective", "insecure", "vulnerable"<br>−: "clean", "perfect", "secure", "invulnerable" | 63.10 |

those in the full-data setting in Table 3, we can find that the model performance shows severe drop. For the CodeT5 model, it even does not converge under the zero-shot and 16-shot settings due to the extreme lack of training data. The low results are reasonable since pre-trained models require task-specific data for better adapting to downstream tasks. However, we observe that with prompt tuning, the pre-trained models achieve significantly better performance than the models using fine-tuning. On average, prompt tuning outperforms fine-tuning by 2.59%, 2.16%, and 2.08% on CodeBERT, CodeT5-small and CodeT5-base, respectively. Note that prompt tuning under zero shot setting even outperforms prompt tuning with 32 shots and fine-tuning with 64 shots. It indicates that the knowledge of pre-trained model can be elicited by the prompt without tuning the parameters.

For the code summarization task, we choose 100, 200, 300, 500, and 1000 training shots as subsets. Figure 4 shows comparison on BLEU-4 scores of prompt tuning and fine-tuning CodeT5 models on different programming languages. We can find that although the model performance drops significantly on the subsets, prompt tuning consistently outperforms fine-tuning, showing an average improvement at 28.08% and 26.86% for CodeT5-small and CodeT5-base, respectively. We also observe that the improvement becomes less stark with the growth of training shots. The results demonstrate that prompt tuning is more advantageous on few training data than fine-tuning.

**Performance in cross-domain scenario.** For some programming languages, the training data are generally insufficient. As shown in Table 1, the data sizes of languages such as Java and Python are greatly larger than those of languages including Javascript and Ruby. Cross-domain learning is one popular solution, i.e., transferring the knowledge of similar domains with sufficient data to the target domains. We use the code summarization task for evaluating the performance of prompt tuning under cross-domain setting. Considering the adequacy of training data and domain similarity, we perform training on the programming language Java or Python, and evaluate on the language with fewer data such as Ruby, JavaScript, and Go. Table 7 shows the cross-domain BLEU-4 scores achieved by CodeT5. We can observe that prompt tuning achieves better performance than fine-tuning for most cross-domain settings, except for the adaption from Python to JavaScript. With prompt tuning, the BLEU-4 scores of CodeT5-small and CodeT5-base are increased by 2.53% and 5.18% on average, respectively.

**Finding 2:** Prompt tuning is more effective in low-resource scenarios than fine-tuning. The fewer training instances, the larger the improvement achieved by prompt tuning. Besides, prompt tuning also shows superior performance on the cross-domain code intelligence task.

## 4.3 RQ3: Impact of Different Prompts

In this RQ, we explore the impact of different prompts on the performance of prompt tuning. We focus on the following three aspects: 1) hard prompt template; 2) hard prompt v.s. vanilla soft prompt; and 3) length of prefix soft prompt.

*4.3.1 Different Hard Prompt Templates.* There are two factors that can impact the performance of hard prompts, including the template design and verbalizer. Due to the space limit, we present the evaluation results on the classification task, i.e., defect detection, and one generation task, i.e., the code summarization. Note that we have the same observation for the code translation task.

**Table 10: Results (BLEU-4 scores) of prompt tuning with different prompt templates on the code summarization task. There is no verbalizer for the prompts of generation tasks.**

| | $f_{prompt}(\cdot)$ | Ruby | JavaScript | Go | Python | Java | PHP | Overall |
|---|---|---|---|---|---|---|---|---|
| CodeT5-small | Summarize [*LANG*] [*X*] [*Z*] | 13.45 | 15.01 | 21.20 | 17.82 | 18.43 | 24.52 | 18.41 |
| | [*SOFT*] * 2 [*X*] [*Z*] | 13.33 | 14.96 | 21.17 | 17.93 | 18.29 | 24.61 | 18.38 |
| | Generate comments for [*LANG*] [*X*] [*Z*] | 13.44 | 14.96 | 21.24 | 17.90 | 18.52 | 24.46 | 18.42 |
| | [*SOFT*] * 4 [*X*] [*Z*] | 13.49 | 14.87 | 21.29 | 17.92 | 18.34 | 24.68 | 18.44 |
| CodeT5-base | Summarize [*LANG*] [*X*] [*Z*] | 13.67 | 15.91 | 22.51 | 18.00 | 19.63 | 25.76 | 19.25 |
| | [*SOFT*] * 2 [*X*] [*Z*] | 13.86 | 15.75 | 22.48 | 18.12 | 19.52 | 25.91 | 19.27 |
| | Generate comments for [*LANG*] [*X*] [*Z*] | 13.68 | 15.84 | 22.49 | 18.03 | 19.59 | 25.88 | 19.25 |
| | [*SOFT*] * 4 [*X*] [*Z*] | 13.74 | 15.82 | 22.63 | 18.06 | 19.60 | 25.83 | 19.28 |

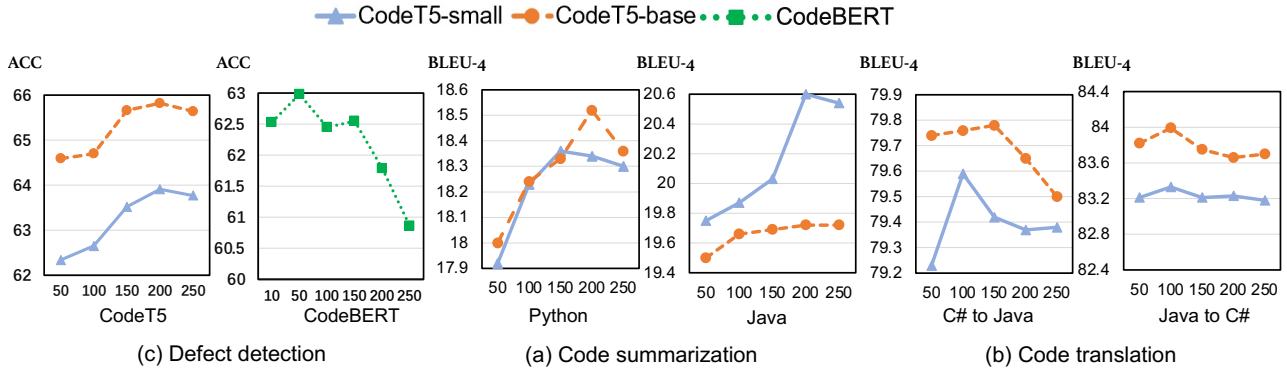


**Figure 5: BLEU-4 score of comparing the performance of CodeT5 models on code summarization and code translation with different prefix lengths. The horizontal axis indicates the length of prefix.**

**Template Design.** The natural language tokens in hard prompt templates are manually defined. To study the impact of different tokens in the template, we conduct experiments with fixed verbalizers. Table 8 and Table 10 show the results on the defect detection task and code summarization task, respectively. Comparing the row 2-5 in Table 8, we can find that the template design impacts the model performance. For example, when using the hard prompt "*The code* [*X*] *is* [*Z*]", CodeBERT outperforms the case when using "*A* [*Z*] *code* [*X*]" by 1.39%. In addition, by changing "*The code* [*X*] *is* [*Z*]" to "[*X*] *The code is* [*Z*]" in which only the token order is different, a drop in performance by 0.8% is observed. However, comparing row 2 and 4 in Table 10, we can find that the model performance is less affected by the template design for the code summarization task. This may be attributed to that only few prompt tokens in the templates can hardly provide helpful guidance for the large solution space in the code summarization task. Thus, we achieve that the template design for hard prompt is more important for the classification task than the generation task.

**Different Verbalizers:** We fix the hard prompt template as " *The code* [*X*] *is* [*Z*]" and analyze the impact of different verbalizers on the model performance. Specifically, we choose task-relevant label words for the verbalizers, with the results on the defect prediction task shown in Table 9. We can observe that different verbalizers influence the performance of prompt tuning. When choosing label words such as *"yes"* and *"no"* (row 2) rather than adjectives to fill the answer slot [*Z*], the result is 0.99% lower than that of using adjectives in the verbalizer (row 3). It indicates that constructing verbalizer with correct grammar is helpful for the prediction. Comparing row 3-6, we can also find that increasing the number of label words is not always beneficial for the model performance, which may be because more label words could introduce bias to the prediction results. When using two label words for indicating each class, the model presents the highest performance.

*4.3.2 Hard Prompt vs. Vanilla Soft Prompt.* As introduced in Section 2.2.2, the vanilla soft prompt replaces the natural language tokens in hard prompt with virtual tokens. The comparison results on the defect detection task are illustrated in Table 8. We experiment with different hard prompts, shown in the first column, with the corresponding vanilla soft prompts at the second column. From the results listed as the last two columns, we can find that hard prompts present better prediction accuracy than the corresponding vanilla soft prompts. For the code summarization task, the results are shown in Table 10. Comparing the performance of hard prompts such as "*Summarize* [*LANG*] [*X*][*Z*]" and "*Generate comments for* [*LANG*] [*X*] [*Z*]" with the corresponding vanilla soft version, we can observe that the difference is marginal, which may be due to the large generation space of the task. Thus, we summarize that hard prompts may be more effective than the corresponding vanilla

```
Turntabler.AuthorizedUser.update_laptop", "original_string": "def update_laptop(name)
      assert_valid_values(name, *%w(mac pc linux chrome iphone cake intel android))
      api('user.modify', :laptop => name)
      self.attributes = {'laptop' => name}
      true
    end", "language": "ruby", "code": "def update_laptop(name)
      assert_valid_values(name, *%w(mac pc linux chrome iphone cake intel android))
      api('user.modify', :laptop => name)
      self.attributes = {'laptop' => name}
      true
    end
```

**(a) Ground truth comment**: Updates the laptop currently being used
**(b) Comment generated by fine-tuning**: Modify the laptop.
**(c) Comment generated by prompt tuning**: Update the laptop.

**Figure 6: Case study on the code summarization task, where the pre-trained model is CodeT5-small.**

```
public virtual bool
contains(object o){
    return indexOf(o) != -1;
}
```

(a) Original C# code

```
public boolean contains(Object o) {
    return indexOf(o) != -1;
}
```

(b) Ground truth Java code

```
public boolean contains(Object o)
{
    return indexOf(o);
}
```

(c) Generated Java code by fine-tuning

```
public boolean contains(Object o) {
    return indexOf(o) != -1;
}
```

(d) Generated Java code by prompt tuning

**Figure 7: Case study on the code translation task, where the pre-trained model is CodeT5-small.**

soft prompts for classification tasks, and the advantage tends to be weakened for generation tasks.

*4.3.3 Different Lengths of Prefix Soft Prompts.* We also study the impact of different lengths of prefix soft prompts. We illustrate the performance under different prefix prompt lengths for the three tasks in Figure 5. As can be seen, too short or long lengths of prefix prompts can degrade the model performance. For all the tasks, prompt tuning achieves the best or nearly best performance when the length of prefix prompt set to a value between 100 and 200. In our work, the prefix lengths are set as 200, 200, and 100 for defect detection, code summarization and code translation tasks, respectively.

**Finding 3:** Prompt templates have large impact on the performance of prompt tuning. It is crucial to construct prompt templates with suitable template design and verbalizers based on domain knowledge. When using the prefix prompts, the length of prompts has impact on the model performance.

# 5 DISCUSSION

## 5.1 Implications of Findings

*Implication on the Utilization of Pre-trained Models.* Prompt tuning performs well in adapting pre-trained models on code intelligence tasks. We observe that prompt tuning can consistently outperform fine-tuning in our experiments under full-data settings, data scarcity settings, and cross-domain settings. The advantage of prompt tuning is especially outstanding in data scarcity settings, which suggests that prompt-tuning is a superior solution when there is a lack of task-specific data.

*Implication on the Utilization of Prompts.* Our experiments demonstrate that different templates and verbalizers influence the performance of the code intelligence tasks. The templates that have the same semantics but different prompt tokens can lead to different performance results. Researchers could try different combinations of the words in their templates and evaluate the effectiveness through experiments. Besides, although the vanilla soft prompt is helpful to reduce the manual cost of prompt template designing, the best performance is achieved mostly by well-designed hard prompt. Furthermore, we find that the performance of prefix soft prompt varies with its length. Determining the best length of the prompt for a downstream task is difficult. Based on our experiments, in general, promising results can be achieved by soft prompt when the length is between 100 and 200.

## 5.2 Case Study

In this section, we provide additional case studies to qualitatively compare prompt tuning with fine-tuning.

The case in Figure 6 shows a Ruby code snippet with comments generated by fine-tuning and prompt tuning models. From the case we can observe that the fine-tuning model is mislead by the word "*modify*" in the code snippet and fails to capture the main functionality "*update*". Quite the opposite, the prompt tuning model accurately summarizes the code snippet.

We also give another case in code translation task in Figure 7. The original C# code (a) is to check whether object *o* is contained. The code translated by fine-tuning model (c) only returns the index of *o* but does not compare it with -1, where the code semantic changes. However, the prompt tuning model generates the identical Java code (d) with the ground truth one (b).

## 5.3 Future Directions

Based on the findings and implications, we suggest two possible future directions for prompt tuning on source code. First, we suggest future research to consider more characteristics of source code, like syntactic structures, in the design of template and the choices of verbalizer. Experiment results demonstrate that domain knowledge plays an important role on the design of prompts. As code structure information has been demonstrated on the design of regular DL models for code-related tasks [11, 14, 31, 32, 58], we believe that the domain knowledge carried by them can also help the design of prompts. Second, through constructing cloze-style prompt template, the factual knowledge and biases contained in the pre-trained models can be investigated [20, 42, 60]. Researchers can focus on improving the interpretability and robustness of pre-trained models and designing novel pre-training tasks in the future.

## 5.4 Threats to Validity

We have identified the following major threats to validity:

**Limited datasets.** The experiment results are based on a limited number datasets for each code intelligence task. The selection of data and datasets may bring bias to the results. To mitigate this issue, we choose the most widely-used datasets for each code-related task, modify the seeds and run the sampling multiple times. We also plan to collect more datasets in the future to better evaluate the effectiveness of prompt tuning.

**Limited downstream tasks.** Our experiments are conducted on three code intelligence tasks, including one classification task and two generation tasks. Although these tasks are the representative ones in code intelligence, there are many other tasks, such as code search [3, 12] and bug fixing [37, 61]. We believe that we could obtain similar observations on these tasks since they can all be formulated as either classification tasks or generation tasks for source code. We will evaluate more tasks with prompt tuning in our future work.

**Suboptimal prompt design.** We demonstrate that prompt tuning can improve the performance of pre-trained models. However, the prompts we use in this paper may not be the best ones. It is challenging to design the best prompt templates and verbalizers, which will be an interesting future work.

# 6 RELATED WORK

## 6.1 Pre-training on Programming Language

Code intelligence aims at learning the semantics of programs to facilitate various program comprehension tasks, such as code search, code summarization, and bug detection [12, 19, 23, 24, 27, 54, 55, 58]. Recently, inspired by the huge success of pre-trained models in NLP, a boom of pre-training models on programming languages arises. CuBERT [21] and CodeBERT [9] are two pioneer works. CuBERT utilizes the MLM pre-training objective in BERT [6] to obtain better representation of source codes. CodeBERT is able to learn NL-PL representation via replaced token detection task [5]. Svyatkovskiy et al. [52] and Kim et al. [22] train GPT-2 [44] on large scale programming languages for solving code completion task. The work GraphCodeBERT [14] leverages data flow graph (DFG) in model pre-training stage, making model better understand the code structure.

Apart from aforementioned encoder or decoder only models, pre-trained models that utilize both encoder and decoder are also proposed for programming languages. For example, Ahmad et al. propose PLBART [1], which is able to support both understanding and generation tasks. The work [8, 37] utilizes text to text transfer transformer (T5) framework to solve code-related tasks. Wang et al. modify the pre-training and finetuning stages of T5 and propose CodeT5 [56].

## 6.2 Prompt Tuning

The concept of prompt tuning is formed gradually. In the work [42], the authors find that the pre-trained language models have ability to learn the factual knowledge due to the mask-and-predict pre-training approach. Therefore, pre-trained language models can be regarded as a kind of knowledge base. To measure the capability of pre-trained models to capture factual information, they propose a language model analysis dataset (LAMA). Later, Jiang et al. attempt to more accurately estimate the knowledge constrained in the language model [20]. They propose LPAQA to automatically discovery better prompt templates. Several works focus on exploring good templates. Yuan et al. [57] replace phases in the template via a thesaurus. The work [17] utilizes a neural prompt rewriter to improve the model performance. Aforementioned works explore the manual templates or hard templates (meaning the words in the template are fixed and not learnable). Researchers also attempt to optimize the template in the training process (soft prompt) [26, 53, 60]. For example, Li et al. add an additional learnable matrix in front of the input embedding [26]. Zhong et al. propose to initialize these matrices by natural language tokens for more effective optimization [60]. Recently, a series of works also study prompts in pre-training stage. They find that the behavior of language models can be manipulated to predict desired outputs [2, 41, 43, 48], sometimes even require no task specific training. In our work, we adapt prompt tuning in code intelligence tasks to exploit knowledge about both natural language and programming languages captured by pre-trained models.

# 7 CONCLUSION

In this paper, we experimentally investigate the effectiveness of prompt tuning on three code intelligence tasks with two pre-trained models. Our study shows that prompt tuning can outperform fine-tuning under full-data settings, data scarcity settings, and cross-domain settings. We summarize our findings and provide implications that can help researchers exploit prompt tuning effectively in their code intelligence tasks. Our source code and experimental data are publicly available at:*https://github.com/adf1178/PT4Code*.

# ACKNOWLEDGMENT

This research was supported by National Natural Science Foundation of China under project No. 62002084, Stable support plan for colleges and universities in Shenzhen under project No. GXWD20201230155427003-20200730101839009, the Major Key Project of PCL (Grant No. PCL2022A03, PCL2021A02, PCL2021A09), Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies (2022B1212010005), and the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14210920 of the General Research Fund).

We would like to express our gratitude to Zijun Yao, who provided valuable advice on our work.

# REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).
[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[3] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 964–974.
[4] Yitian Chai, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Cross-Domain Deep Code Search with Few-Shot Meta Learning. *arXiv preprint arXiv:2201.00150* (2022).
[5] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2019. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In *International Conference on Learning Representations*.
[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[7] Ning Ding, Shengding Hu, Weilin Zhao, Yulin Chen, Zhiyuan Liu, Hai-Tao Zheng, and Maosong Sun. 2021. Openprompt: An open-source framework for prompt-learning. *arXiv preprint arXiv:2111.01998* (2021).
[8] Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. CodeTrans: Towards Cracking the Language of Silicon's Code Through Self-Supervised Deep Learning and High Performance Computing. *arXiv preprint arXiv:2104.02443* (2021).

[9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[10] Wei Fu and Tim Menzies. 2017. Easy over hard: A case study on deep learning. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 49–60.

[11] Wenchao Gu, Zongjie Li, Cuiyun Gao, Chaozheng Wang, Hongyu Zhang, Zenglin Xu, and Michael R Lyu. 2021. CRaDLe: Deep code retrieval based on semantic dependency learning. *Neural Networks* 141 (2021), 385–394.

[12] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

[13] Yuxian Gu, Xu Han, Zhiyuan Liu, and Minlie Huang. 2021. Ppt: Pre-trained prompt tuning for few-shot learning. *arXiv preprint arXiv:2109.04332* (2021).

[14] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[15] Yunhui Guo, Honghui Shi, Abhishek Kumar, Kristen Grauman, Tajana Rosing, and Rogerio Feris. 2019. Spottune: transfer learning through adaptive fine-tuning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 4805–4814.

[16] Xu Han, Weilin Zhao, Ning Ding, Zhiyuan Liu, and Maosong Sun. 2021. Ptr: Prompt tuning with rules for text classification. *arXiv preprint arXiv:2105.11259* (2021).

[17] Adi Haviv, Jonathan Berant, and Amir Globerson. 2021. BERTese: Learning to Speak to BERT. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*. 3618–3623.

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

[19] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.

[20] Zhengbao Jiang, Frank F Xu, Jun Araki, and Graham Neubig. 2020. How can we know what language models know? *Transactions of the Association for Computational Linguistics* 8 (2020), 423–438.

[21] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*. PMLR, 5110–5121.

[22] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 150–162.

[23] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 218–229.

[24] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension*. 184–195.

[25] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691* (2021).

[26] Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190* (2021).

[27] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[28] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. 2021. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing* (2021).

[29] Pengfei Liu, Xipeng Qiu, and Xuanjing Huang. 2016. Recurrent neural network for text classification with multi-task learning. *arXiv preprint arXiv:1605.05101* (2016).

[30] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2021. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *arXiv preprint arXiv:2107.13586* (2021).

[31] Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2020. Retrieval-Augmented Generation for Code Summarization via Hybrid GNN. In *International Conference on Learning Representations*.

[32] Shangqing Liu, Cuiyun Gao, Sen Chen, Nie Lun Yiu, and Yang Liu. 2020. ATOM: Commit message generation based on abstract syntax tree and hybrid ranking. *IEEE Transactions on Software Engineering* (2020).

[33] Xiao Liu, Kaixuan Ji, Yicheng Fu, Zhengxiao Du, Zhilin Yang, and Jie Tang. 2021. P-Tuning v2: Prompt Tuning Can Be Comparable to Fine-tuning Universally Across Scales and Tasks. *arXiv preprint arXiv:2110.07602* (2021).

[34] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[35] Ilya Loshchilov and Frank Hutter. 2018. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*.

[36] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021).

[37] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. 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)*. IEEE, 336–347.

[38] Sewon Min, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. 2021. Noisy channel language model prompting for few-shot text classification. *arXiv preprint arXiv:2108.04106* (2021).

[39] Sinno Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering* 22, 10 (2009), 1345–1359.

[40] Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *Proceedings of NAACL-HLT*. 2227–2237.

[41] Fabio Petroni, Patrick Lewis, Aleksandra Piktus, Tim Rocktäschel, Yuxiang Wu, Alexander H Miller, and Sebastian Riedel. 2020. How Context Affects Language Models' Factual Predictions. *arXiv preprint arXiv:2005.04611* (2020).

[42] Fabio Petroni, Tim Rocktäschel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, Alexander H Miller, and Sebastian Riedel. 2019. Language models as knowledge bases? *arXiv preprint arXiv:1909.01066* (2019).

[43] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

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

[45] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683* (2019).

[46] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).

[47] Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *Advances in Neural Information Processing Systems* 33 (2020), 20601–20611.

[48] Timo Schick and Hinrich Schütze. 2020. It's not just size that matters: Small language models are also few-shot learners. *arXiv preprint arXiv:2009.07118* (2020).

[49] Timo Schick and Hinrich Schütze. 2021. Exploiting Cloze-Questions for Few-Shot Text Classification and Natural Language Inference. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*. 255–269.

[50] Zhiqiang Shen, Zechun Liu, Jie Qin, Marios Savvides, and Kwang-Ting Cheng. 2021. Partial is better than all: Revisiting fine-tuning strategy for few-shot learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 9594–9602.

[51] Zhensu Sun, Li Li, Yan Liu, and Xiaoning Du. 2022. On the Importance of Building High-quality Training Datasets for Neural Code Search. *arXiv preprint arXiv:2202.06649* (2022).

[52] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. 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*. 1433–1443.

[53] Maria Tsimpoukelli, Jacob Menick, Serkan Cabi, SM Eslami, Oriol Vinyals, and Felix Hill. 2021. Multimodal few-shot learning with frozen language models. *Advances in Neural Information Processing Systems* 34 (2021).

[54] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 397–407.

[55] Wenhan Wang, Ge Li, Sijie Shen, Xin Xia, and Zhi Jin. 2020. Modular Tree Network for Source Code Representation Learning. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–23.

[56] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[57] Weizhe Yuan, Graham Neubig, and Pengfei Liu. 2021. Bartscore: Evaluating generated text as text generation. *Advances in Neural Information Processing*

*Systems* 34 (2021).

[58] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[59] Ningyu Zhang, Luoqiu Li, Xiang Chen, Shumin Deng, Zhen Bi, Chuanqi Tan, Fei Huang, and Huajun Chen. 2021. Differentiable prompt makes pre-trained language models better few-shot learners. *arXiv preprint arXiv:2108.13161* (2021).

[60] Zexuan Zhong, Dan Friedman, and Danqi Chen. 2021. Factual probing is [mask]: Learning vs. learning to recall. *arXiv preprint arXiv:2104.05240* (2021).

[61] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 14–24.

[62] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).