Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models

Martin Weyssow
DIRO, University of Montreal
Montreal, Canada
martin.weyssow@umontreal.ca

Xin Zhou
Singapore Management University
Singapore
xinzhou.2020@phdcs.smu.edu.sg

Kisub Kim
Singapore Management University
Singapore
kisubkim@smu.edu.sg

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

Houari Sahraoui
DIRO, University of Montreal
Montreal, Canada
sahraouh@iro.umontreal.ca

ABSTRACT

Large Language Models (LLMs) possess impressive capabilities to generate meaningful code snippets given natural language intents in zero-shot, *i.e.*, without the need for specific fine-tuning. In the perspective of unleashing their full potential, prior work has demonstrated the benefits of fine-tuning the models to task-specific data. However, fine-tuning process demands heavy computational costs and is intractable when resources are scarce, especially for models with billions of parameters. In light of these challenges, previous studies explored In-Context Learning (ICL) as an effective strategy to generate contextually appropriate code without fine-tuning. However, it operates at inference time and does not involve learning task-specific parameters, potentially limiting the model's performance on downstream tasks. In this context, we foresee that Parameter-Efficient Fine-Tuning (PEFT) techniques carry a high potential for efficiently specializing LLMs to task-specific data. In this paper, we deliver a comprehensive study of LLMs with the impact of PEFT techniques under the automated code generation scenario. Our experimental results reveal the superiority and potential of such techniques over ICL on a wide range of LLMs in reducing the computational burden and improving performance. Therefore, the study opens opportunities for broader applications of PEFT in software engineering scenarios.

ACM Reference Format:
Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. 2024. Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Large Language Models (LLMs) based on the Transformer architecture [62] showcase substantial potential across diverse domains such as natural language processing (NLP) [28, 40, 70], computer vision [9, 53, 78], and software engineering [10, 61, 75]. These models possess a high capability to comprehend natural language intents in zero-shot scenarios, meaning they can perform tasks without the need for specific fine-tuning. As a result, they have garnered significant interest in the software engineering field for automating code-related tasks such as program repair [26, 73, 74] and code generation [4, 10, 44].

While the zero-shot capabilities of LLMs are impressive, they often achieve peak effectiveness and show their full potential when subjected to fine-tuning [49, 69]. Specifically, adapting an LLM to task-specific data allows it to further improves its comprehension of the potentially highly contextual data at hand and thus provide more meaningful generated content. However, this process comes at a significant computational cost. Full fine-tuning, where all the parameters of the LLMs are updated during training, demands remarkable computational resources, especially when the LLM contains billions of parameters [57]. In response to the challenges posed by the computationally expensive full fine-tuning of LLMs, prior studies [48, 73, 87] in software engineering have explored alternative approaches. These approaches aim to mitigate the computational burden while maintaining tolerable effectiveness on downstream tasks. One such technique gaining traction is In-Context Learning (ICL) [7, 49]. ICL is a technique that consists of providing prompt examples of the task to the LLM, guiding it to generate contextually appropriate content without any fine-tuning involved. This technique has already shown promising results for code-related tasks, including automated program repair [73], bug fixing [48], and code generation [55, 67, 87].

Although ICL provides a viable alternative to full fine-tuning, it operates at inference time, and it neither allows learning nor updating any parameters, which may prevent capturing more fine-grained information about the task. It can result in a potential loss of effectiveness. In this context, Parameter-Efficient Fine-Tuning (PEFT) techniques have emerged as promising solutions to render the fine-tuning cost at the lowest while allowing the model to learn task-specific parameters. Prior works [11, 64, 65] in code intelligence have demonstrated the capability of PEFT techniques, and often shown their superiority over full fine-tuning across a wide range of tasks. However, these studies focus on small language models (<0.25B parameters) such as CodeBERT [15] and CodeT5 [66] and overlooked the applicability of PEFT techniques to

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

LLMs (≥1B parameters), leaving an important research gap. Given the growing ubiquity of LLMs, we believe addressing this gap is of paramount importance in advancing the field of code intelligence and harnessing the full potential of LLMs. Furthermore, we identify an additional research opportunity in exploring the usage of PEFT techniques under limited resource scenarios, aiming to demonstrate the democratization of LLMs tuning through PEFT. Addressing these gaps will not only show how PEFT techniques can enhance the effectiveness of LLMs but also how they allow to broaden the accessibility and utility of LLMs in scarce computation settings.

In this paper, we delve into the realm of PEFT techniques for LLMs. We focus our study on code generation, which has been a pivotal area of research due to its transformative impact on automating software development [10, 44, 46]. Our objective is twofold: (1) to assess the code generation capability of LLMs with ICL and two representative PEFT techniques and (2) to study in-depth the effectiveness of PEFT techniques. Additionally, we conduct our comparative study with limited availability of computational resources to investigate the broad practicality of using PEFT techniques for LLMs. To achieve these objectives, we formulate three research questions that guide our study:

- RQ1: How do LLMs tuned using PEFT perform on code generation compared to small language models?
- RQ2: Are PEFT techniques more promising than ICL for LLMs of code?
- RQ3: What are the reasons behind the improvement of LLMs tuned with PEFT techniques compared to LLMs with ICL?

Answering RQ1 and RQ2 fulfills our first objective by providing key insights regarding PEFT techniques. Specifically, we gain a comprehensive understanding of how LLMs tuned with PEFT techniques compare to fully fine-tuned small language models and LLMs with ICL. Then, by addressing RQ3, we achieve our second objective, wherein we conduct a detailed analysis of the disparities between code generated by LLMs with PEFT techniques and ICL.

To address these RQs, we conduct experiments on the CoNaLa dataset [79] for Python code generation using three representative metrics: Exact Match@k (EM@k), BLEU, and CodeBLEU. Conversely to evaluation datasets such as HumanEval [10], the CoNaLa dataset, widely used in prior code generation studies [45, 67, 87], includes training examples that can be employed for fine-tuning. For a comprehensive comparative study, we select four distinct model families: PolyCoder [75], Bloom [52], CodeGen [44] and InCoder [16], encompassing a variety of versions, including seven large and three small variants. Note that we omitted closed-sourced LLMs such as Codex due to the inaccessibility of their parameters, which makes the study of any fine-tuning technique infeasible. Furthermore, we incorporate two PEFT techniques: LoRA [23] and Prompt tuning [29]. Unlike ICL, these techniques entail learning new parameters to tune the LLMs for the specific downstream task. Our main findings are the followings:

- LLMs fine-tuned with PEFT techniques, *i.e.,* a few millions of parameters, systematically outperform small language models fully fine-tuned with hundreds of millions of parameters.
- Prompt tuning often outperforms LoRA even though it requires learning substantially fewer parameters.
- LLMs fine-tuned using LoRA and Prompt tuning significantly outperform LLMs with ICL, even when increasing the number of prompt examples under the ICL setting.
- PEFT techniques allow LLMs to better adapt to the task-specific dataset with low computational cost. Furthermore, we demonstrate the usability of PEFT techniques on a single 24GB VRAM GPU for models up to 7B parameters.

Our study sheds light on the promising opportunities that PEFT techniques hold, warranting further exploration for their application in other code-related tasks and scenarios. To summarize our contributions are the followings:

- We conduct the first comprehensive effectiveness exploration of two state-of-the-art PEFT techniques for Python code generation over a broad range of LLMs. We study Prompt tuning and LoRA, with the latter being unexplored in the field of code intelligence and both unexplored for tuning LLMs for code generation.
- A comprehensive comparison and analysis of PEFT techniques and ICL for LLMs on code generation.
- We demonstrate the practicality of PEFT techniques and ICL to effectively reduce the computational burden when fine-tuning LLMs, showcasing their potential broader applications to software engineering scenarios.

2 BACKGROUND

We introduce overall categories and representative techniques that we study in this paper to support a comprehensive understanding.

2.1 Large Language Models (LLMs)

LLMs have garnered considerable interest from researchers, practitioners, and the wider community due to their impressive capabilities and potential. Their capacity to comprehend and generate text that closely resembles human language has sparked significant curiosity across diverse fields such as natural language processing (NLP) [28, 40, 70], computer vision [82, 86], and software engineering [10, 51, 61]. The profound impact of LLMs continues to drive advancements in language understanding, generation, and a wide range of applications across various research fields and industries [27, 56, 59]. "LLM" often refers to language models containing billions of parameters. In this study, we consider LLMs as models having more than 1B parameters, and small language models for those having less than 1B parameters.

Various LLMs. The utilization and capacity of LLMs vary based on the characteristics [84] of their training data and the number of parameters. On the one hand, representative LLMs for NLP such as GPT [46], OPT [83], and Bloom [52] are trained based on diverse and extensive datasets sourced from various domains, including books, articles, websites, and other textual sources. LLMs for code such as Codex [10], CodeGen [44], InCoder [16], and AlphaCode [32] are trained on vast source code repositories. They have also demonstrated remarkable progress on the tasks related to source code such as program synthesis [4, 25], code summarization [1], and program repair [26, 73, 74]. On the other hand, larger models with a higher number of parameters tend to possess enhanced understanding and generation abilities [75].

In-Context Learning (ICL) for LLMs. As one of the specific types of LLM-related techniques, ICL has emerged as an effective technique [7, 12, 33, 41, 47]. ICL seeks to improve the abilities of LLMs by integrating context-specific information, in the form of an input prompt or instruction template (see Table 4), during the inference and thus without the need of performing gradient-based training. Therefore, by considering the context, the model becomes more capable of producing coherent and contextually relevant outputs. This contextual coherence of the LLM and not having to perform costly gradient-based training constitutes prime advantages of using ICL to specialize LLMs to a specific task or dataset. However, ICL also presents some inconveniences, including the need to design representative prompts. Another limitation concerns the introduction of the extra prompt input tokens to the model, which may be infeasible when the contextual information is too large.

2.2 Parameter-Efficient Fine-Tuning (PEFT)

PEFT refers to a technique that aims to optimize the fine-tuning process of LLMs by selectively updating a subset of parameters instead of updating the entire model. Fine-tuning language models typically require a large amount of task-specific data, which can be expensive and time-consuming to acquire. Additionally, traditional fine-tuning involves updating all the parameters, which can be computationally expensive, especially when the target model is an LLM. PEFT is a new and recent technique, designed to improve the efficiency of computation, data, and other resources. Thus, it is not only beneficial in scenarios where collecting large amounts of task-specific data is challenging or costly but also in situations where the process has to be faster and more efficient.

Technically, PEFT techniques focus on learning a small number of parameters for the task at hand by designing additional layers [22], adding prepending additional tokens [29, 31], and decomposing weight gradients into specific matrices [23]. One of the representative cutting-edge PEFT techniques is LOw-Rank Adaptation of LLMs (LoRA) [23]. The technique consists of freezing the model weights and injecting low-rank trainable matrices into each layer of the Transformer architecture, thereby drastically reducing the number of trainable parameters. We employ LoRA as one of our PEFT techniques since it has been widely used in NLP [13, 37, 60] and showed promising performance. In addition to LoRA, we also include Prompt tuning [29] as a second PEFT technique. Prompt tuning involves the process of prepending virtual tokens to the input tokens of the LLM. These virtual tokens are differentiable, allowing them to be learned through backpropagation during fine-tuning, while the rest of the LLM remains frozen. The simplicity of its design and its competitive performance with other PEFT techniques [38] make Prompt tuning a valuable addition to our study.

3 EFFICIENT FINE-TUNING OF LLMS WITH LIMITED RESOURCES

In the era of LLMs, the availability of substantial computational resources plays a crucial role in harnessing their high capabilities. Unfortunately, many researchers and practitioners often find themselves constrained by the limited availability of high-end computing infrastructures.

Table 1: Computation-effectiveness trade-off for each model tuning technique.

Technique	Computation costs	Effectiveness
Full fine-tuning	high [X]	high [✓]
ICL	low [✓]	low [X]
PEFT	low [✓]	high [✓]

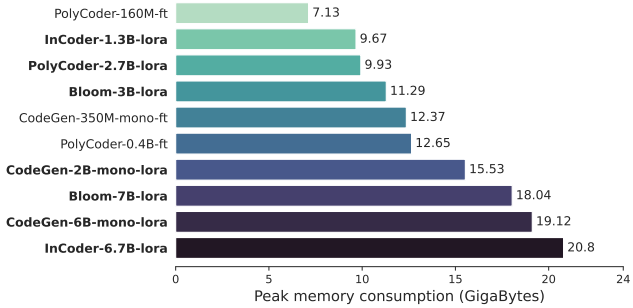

Fig. 1: Peak GPU memory consumption during the fine-tuning of models using full fine-tuning (ft) and LoRA.

Let us assume a scenario where a software engineering practitioner aims to fine-tune LLMs with access limited to only a single consumer GPU with 24GB of VRAM. Such limitations in computational power pose significant challenges when employing traditional full fine-tuning approaches that demand extensive memory resources for fine-tuning LLMs [4, 14, 47]. As LLMs tend to grow in size, the usage of full fine-tuning becomes even more impractical as the increase in the number of trainable parameters requires additional memory. In other words, although full fine-tuning may yield optimal effectiveness on downstream tasks, its application in practice is hindered by its heavy computational costs, underscoring a *computation-effectiveness trade-off* (see Table 1). In this context, an alternative approach consists of leveraging ICL, which has been widely prioritized over full fine-tuning for LLMs in prior studies [12, 33, 41, 48, 73]. This approach renders the computation costs minimal as it does not require any parameter update but generally comes at the cost of lower effectiveness on downstream tasks. ICL also presents notable limitations, such as the selection of representative contextual examples being a challenging endeavor [37]. For instance, the wording of the input prompt and the ordering of the prompt examples can have a significant impact on the model's effectiveness on the downstream task [68, 85]. Altogether, these concerns hold particular significance in software engineering due to the prevalence of highly contextual tasks and the need for effective model specialization to specific datasets.

To overcome these limitations, we foresee the emergence of PEFT techniques as promising solutions, offering more computationally efficient and scalable approaches to fine-tuning LLMs. These techniques constitute a viable alternative that can empower the software engineering practitioner to overcome the challenges posed by the scarcity of computational power and the limitations of ICL. In contrast to full fine-tuning, PEFT alleviates the need for high computational power by reducing the number of parameters optimized at training while being competitive with full fine-tuning

on downstream tasks. As highlighted in Table 1, PEFT techniques could potentially be the most practical solution at the sweet point of the computation-effectiveness trade-off. In this resource-constraint scenario, the software engineering practitioner can effectively fine-tune LLMs without encountering out-of-memory errors by utilizing PEFT techniques such as LoRA (see Fig. 1). Thanks to PEFT techniques, the practitioner can fine-tune LLMs such as InCoder-6.7B containing 6.7B parameters, without utilizing more than 21GB of GPU memory. Moreover, we also observe that the memory consumption when fully fine-tuning PolyCoder-160M and CodeGen-350M-mono is about 7GB and 12GB, respectively. This observation highlights that the memory consumption almost doubles as we only introduce 200M additional parameters. Consequently, we can expect the memory consumption to skyrocket with LLMs [57] with billions of parameters, showcasing the intractability of full fine-tuning for larger models. In conclusion, by leveraging PEFT, software engineering practitioners and researchers can effectively tackle highly contextual tasks, specialize LLMs to specific datasets, and advance research and practice even under limited resources.

4 METHODOLOGY

In this section, we go through the experimental setup of our study. We conduct all the experiments under a resource-constrained scenario. Specifically, all the procedures (*i.e.*, fine-tuning and inference) of the models are performed with access to a single 24GB VRAM GPU. The main objective of our study is to demonstrate whether the fine-tuning of LLMs through PEFT techniques is feasible and desirable over previous approaches and smaller models in this context.

4.1 Research Questions

In this study, we focus on the following research questions:

- RQ1: **How do LLMs tuned using PEFT perform on code generation compared to small language models?** We study the effectiveness of a large spectrum of small language models and LLMs for code generation. We compare how the full fine-tuning of small language models performs against LLMs fine-tuned with PEFT techniques. We select a wide range of models of various sizes, pre-trained on diverse codebases and with different learning objectives to study how these factors impact their effectiveness.
- RQ2: **Are PEFT techniques more promising than ICL for LLMs of code?** We compare LLMs with two representative PEFT techniques, LoRA and Prompt tuning against those with ICL and full fine-tuning. We test all the LLMs using ICL with a number of prompt examples ranging from zero to five. We select prompt examples to cover diverse third-party libraries and standard Python modules.
- RQ3: **What are the reasons behind the improvement of LLMs tuned with PEFT techniques compared to LLMs with ICL?** We select the best-performing LLM from previous RQs and analyze the results more in-depth. We start by exploring how both approaches perform on domain-specific test samples, *i.e.*, involving specific third-party libraries or Python modules. We conclude by assessing the variations in code generation between LLMs that undergo fine-tuning and those that rely solely on ICL examples. This analysis is conducted across three specific examples.

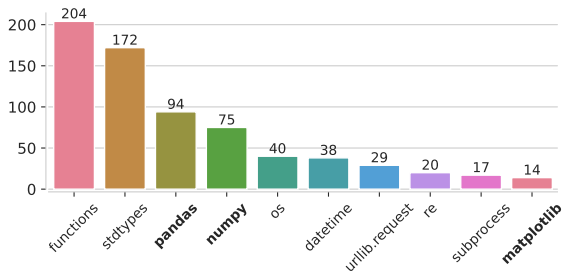

Fig. 2: Distribution of the ten most common Python modules and third-party libraries (bold) in the CoNaLa test dataset.

4.2 Dataset and Task

Throughout our study, we compare all the studied models on a Python code generation task. This task has gained significant attention in recent years [10, 11, 44, 46, 55] with the emergence of LLMs and their capability to generate Python code in zero-shot, *i.e.*, without further fine-tuning. In particular, evaluation datasets such as HumanEval [10] have extensively been used to benchmark code generation approaches [4, 10, 75]. However, HumanEval does not include a training corpus to evaluate fine-tuning approaches. Since the objective of our study is to specialize LLMs using PEFT techniques, we selected the CoNaLa dataset [80]. This dataset provides an ample number of examples that can be employed for fine-tuning a model and has been used in prior code generation studies with LLMs [67, 87].

CoNaLa dataset. We use a curated version of the CoNaLa dataset [87]. The dataset was crawled from StackOverflow and contains manually annotated pairs of code and natural language intent. Each natural language intent contains hints about the manipulated variables in the ground truth code, *e.g.*, see bold text in Table 2. In this curated version of the dataset, the authors ensured that each sample in the validation and test sets contain at least one Python function that does not appear in the training set. Additionally, they ensured that examples crawled from the same StackOverflow post appear in different sets. Thus, we can guarantee that each natural intent in the test does not appear in the training set. Finally, the dataset includes the third-party and Python modules used in each sample, allowing us to analyze in-depth the generation capabilities of each tuning technique (*see RQ2 and RQ3*). The dataset contains 2,135/201/543 samples as the training/validation/test sets, respectively. In Fig. 2, we report the distribution of the ten most common Python modules and third-party libraries in the test set. "functions" refers to standard library functions and "stdtypes" to built-in types functions, while the other module names are explicit. We observe that "pandas" and "numpy" accounts for most of the third-party library samples.

Task design. In Table 2, we illustrate an overview of the task design. The prompt is in the form of an instruction template, where "### Instruction:" and "### Answer:" play the role of delimiting the instruction, *i.e.*, natural language intent, and the answer, *i.e.*, code

Table 2: Overview of the code generation task. The model attempts to generate the ground truth given an instruction template (prompt).

Prompt:	### Instruction: **map two lists 'keys' and 'values' into a dictionary** ### Answer:
Ground truth:	`dict([(k, v) for k, v in zip(keys, values)])`

generation. Note that this prompt design may not be optimal, but this kind of instruction template has shown to be effective in prior works [34, 81]. The code generated by the model is compared with the ground truth to assess the quality of the generation. During fine-tuning, we minimize a standard autoregressive cross-entropy loss function:

$$\mathcal{L} = -\sum_{i=1}^{T+1} M_i \cdot \log P(x_i \mid x_{<i}) \,,$$

where:

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

The model receives a concatenation of the prompt and the ground truth as input and predicts each token x_i in an autoregressive manner given the previous tokens $x_{<i}$. Note that in the computation of the loss, we ignore the tokens from the instruction template to force the model to focus on generating code. We set the value of the instruction tokens to -100 and ignore them in the loss computation using the indicator function M_i. At inference, the model receives the prompt as input and attempts to generate the ground truth code. In our experiments, the model typically outputs a list of $n = 10$ code candidates.

4.3 Small and Large Language Models

In order to enable our study to carry out a comprehensive analysis, we selected our small language models and LLMs according to several criteria. First, we exclusively considered open-source models. We omitted closed-sourced LLMs such as Codex due to the inaccessibility of their parameters, which makes the study of any fine-tuning technique infeasible. All the studied models' checkpoints can be freely accessed, and most of them have been pre-trained using open-source data. Secondly, we selected state-of-the-art LLMs, which have been released within the past year. Additionally, we included one family of models not exclusively pre-trained on code data, allowing for an additional point of comparison. Finally, to investigate the impact of scaling, we selected models with a diverse range of parameters. We consider models with less than 1B parameters as small language models, and the others as LLMs. Table 3 lists the four families of selected LLMs. The type column refers to the type of learning objective used for pre-training. In total, we consider ten models to conduct our experiments.

- **PolyCoder** [75] models are based on the GPT-2 [49] architecture. The models were all pre-trained on snapshots of the most popular GitHub repositories. The pre-training data has about 250GB of data and includes 12 popular programming languages. We include PolyCoder-160M and PolyCoder-0.4B as small language models and PolyCoder-2.7B as an LLM.
- **Bloom** [52] is a family of models leveraging the Megatron-LM GPT2 [54] architecture. Bloom's pre-training data contains 1.6T of preprocessed text and spans 46 natural languages and 13 programming languages. In our study, we select the original checkpoints of Bloom-3B and Bloom-7B. We do not consider Bloom-176B as it would not fit our hardware configuration.
- **CodeGen** [44] models are based on the GPT-Neo (350M, 2.7B) [6] and GPT-J (6B) [63] architectures for program synthesis. The authors released three CodeGen models (NL, multi and mono) that use different pre-training data variants. CodeGen-NL models are pre-trained on the Pile dataset (825GB). CodeGen-Multi models are initialized with CodeGen-NL and further fine-tuned on a dataset of multiple programming languages (BigQuery). CodeGen-Mono models are initialized with CodeGen-Multi and further fine-tuned on Python code (BigPython) We selected CodeGen-350M-mono as one small language models, CodeGen-2B-Mono and CodeGen-6B-Mono as LLMs. We conjecture that the CodeGen-mono family can outperform the other models in our experimental context as they were further fine-tuned on Python data.
- **InCoder** [16] models are based on XGLM [35]. InCoder models were pre-trained using an infilling learning objective consisting of masking spans of contiguous tokens and learning to reconstruct the masked tokens. The pre-training data contains 159GB of code, of which 52GB are in Python, collected from GitHub and StackOverflow. We employ both InCoder-1.3B and InCoder-6.7B in our experiments.

Table 3: Families of LLMs included in our study.

Model	# Parameters	Pre-training data	Type
PolyCoder	160M/0.4B/2.7B	Github	Causal LM
Bloom	3B/7B	ROOTS	Causal LM
CodeGen	350M/2B/3B	The Pile / BigQuery / BigPython	Causal LM
InCoder	1.3B/6.7B	–	Infilling

Table 4: List of examples used for ICL.

Instruction 1:	Create list 'listy' containing 3 empty lists
Solution 1:	`listy = [[] for i in range(3)]`
Instruction 2:	Solve for the least squares' solution of matrices 'a' and 'b'
Solution 2:	`np.linalg.solve(np.dot(a.T, a), np.dot(a.T, b))`
Instruction 3:	Drop a single subcolumn 'a' in column 'col1' from a dataframe 'df'
Solution 3:	`df.drop(('col1', 'a'), axis=1)`
Instruction 4:	remove tags from a string 'mystring'
Solution 4:	`re.sub('<[^>]*>', '', mystring)`
Instruction 5:	convert a python dictionary 'a' to a list of tuples
Solution 5:	`[(k, v) for k, v in a.items()]`

4.4 Fine-Tuning Techniques

We study four previously mentioned fine-tuning techniques, *i.e.,* Full fine-tuning, ICL, LoRA, and Prompt tuning. For each PEFT technique, we sweep hyperparameters that empirically showed having the greatest influence on the LLMs' effectiveness on downstream tasks from their initial papers.

Full fine-tuning. Under our resource-constrained scenario, we were only able to fine-tune the small language models, *i.e.*, PolyCoder-160M, PolyCoder-0.4B, and CodeGen-350M-mono, using full fine-tuning. We did not fully fine-tune the LLMs as it produces out-of-memory errors under our limited resources scenario.

ICL. We consider ICL as a highly efficient way of specializing an LLM to new data as this method does not require any fine-tuning and updating of the parameters of the model. We study ICL by varying n, the number of prompt examples, where $n \in \{0, 1, 2, 3, 4, 5\}$. In Table 4, we list the examples used for ICL from the training set. We use the instruction template in Table 2 and concatenate the templates when $n \geq 2$. Additionally, we ensure the selection of examples covering various modules and libraries, including *pandas*, *numpy*, and *regex*.

LoRA. This PEFT technique makes the fine-tuning efficient by reducing the parameters of the LLM to two update matrices by low-rank decomposition. For each LLM, we tune the r and α hyperparameters, where r is the rank of the update matrices and α is a scaling factor that helps stabilize the training. We experiment with $r, \alpha \in \{(8, 16), (8, 32), (16, 32), (16, 64), (32, 64)\}$. We select r and α following values used in the original paper of LoRA [23] and apply LoRA to the attention layers of the LLMs.

Prompt tuning. We prepend a set of n continuous virtual tokens to each input sample of the LLM. The parameters associated with the virtual tokens are updated, while the rest of the model's parameters are kept frozen during fine-tuning. We study Prompt tuning by varying n, the number of virtual tokens, with $n \in \{5, 10, 20, 40, 100, 200\}$. Similarly to LoRA, we select n according to values used in the original paper of Prompt tuning [29].

4.5 Metrics

We measure the effectiveness of the models through widely used metrics in prior code generation work. To evaluate each prediction, we report the Exact Match (EM), BLEU, and CodeBLEU [50] metrics. At inference, we generate 10 candidates per test sample. Therefore, to evaluate the effectiveness of the models on a list of $k \in [1, 10]$ candidates, we report the EM@k, which computes the average correct predictions among a list of k candidates.

4.6 Implementation Details

We used a single NVIDIA GeForce RTX 3090 with 24GB of VRAM. We leveraged the HuggingFace library [72] to implement the models and for fine-tuning and test. Note that we did not experiment Prompt tuning with InCoder models as the implementation of Prompt tuning presents some issues in Huggingface library. In terms of hyperparameters, we found that LoRA and Prompt tuning work better with higher learning rates, *i.e.*, $5e-5$ for full fine-tuning, $3e-4$ for LoRA and $3e-3$ for Prompt tuning. We used half-precision for the LLMs to fit the models in our GPU, and full precision for the small language models. At inference, we found that beam search with a high temperature works better than alternative decoding strategies such as nucleus sampling [21]. We report all the hyperparameters in our replication package and make all the data and models available: https://anonymous.4open.science/r/peft-llm-code-251E/

5 EXPERIMENTAL RESULTS

5.1 RQ1: Performance Exploration Based on Model Size and PEFT Application

We report the effectiveness of the small language models, LLMs with PEFT applications on code generation in Table 5. Note that due to limited resources, we were not able to fine-tune the LLMs using full fine-tuning.

Effectiveness of small language models vs LLMs. Among the small language models, we observe that CodeGen-350M-mono shows the highest effectiveness across all metrics. The gap with PolyCoder-0.4B likely originates from the fact that CodeGen-350M-mono was pre-trained on more data and further fine-tuned on Python code. Our observations align with prior studies [44, 67, 87] that identified CodeGen-350M-mono as a robust small language model for the Python code generation task. Regarding LLMs, all models outperform the small language models, except for PolyCoder-2.7B, which shows comparable performance to CodeGen-350M-mono. This result highlights the relevance of exploiting PEFT techniques, as they enable fine-tuning LLMs with limited resources.

We also report the number of parameters required to fine-tune each model in Table 5. Both PEFT techniques drastically reduce the number of parameters while outperforming the small language models fine-tuned with hundreds of millions of parameters. As expected, CodeGen-6B-mono emerges as the best-performing LLM, and CodeGen-2B-mono competes with InCoder-6.7B despite its base model, *i.e.*, prior to applying PEFT, being substantially lighter. Regarding Bloom models, their performance falls short compared to CodeGen and InCoder models, underscoring the significance of pre-training models on a larger code dataset or performing further fine-tuning. Furthermore, we have noticed considerable disparities in the EM@k metrics, while smaller gaps are observed in BLEU and CodeBLEU. We attribute the lower gaps in CodeBLEU to the metric's reliance on complex structures like data flow, which are scarce in the short test samples from the CoNaLa dataset. Consequently, we focus the comparison on the EM@k metrics for the remainder of this study.

LoRA vs Prompt tuning. For each LLM, we do not observe a significant difference between LoRA and Prompt tuning, except for PolyCoder-2.7B. We highlight the best model in bold by selecting the PEFT technique that outperforms the other in most evaluation metrics.

> **Answer to RQ1**: In the context of limited computation resources, LLMs consistently outperform small language models. CodeGen-6B-mono tuned with Prompt tuning is our best-performing LLM, and CodeGen-2B-mono competes with larger LLMs such as InCoder-6.7B. CodeGen-6B-mono-pt improves each EM@k metric by 95%–117% compared to the best-performing small language model, *i.e.*, CodeGen-350M-mono.

Table 5: [RQ1] – Comparison of the effectiveness of the small language models fully fine-tuned (*best underlined*) and LLMs fine-tuned using PEFT techniques (*best in bold*).

	Model	# Parameters	EM@1	EM@2	EM@5	EM@10	BLEU	CodeBLEU
Small LMs	PolyCoder-160M	160M	3.31	5.16	8.47	10.13	7.11	16.42
	PolyCoder-0.4B	400M	3.50	6.45	10.68	12.15	6.95	14.51
	CodeGen-350M-mono	350M	7.37	9.76	15.10	17.68	9.78	17.89
LLMs	**PolyCoder-2.7B-lora**	10.4M (0.38%)	7.73	10.50	14.73	16.02	9.83	17.80
	PolyCoder-2.7B-pt	0.2M (0.01%)	5.34	7.75	10.50	12.89	5.35	14.39
	Bloom-3B-lora	9.8M (0.33%)	9.58	13.81	19.15	23.05	9.70	16.82
	Bloom-3B-pt	0.5M (0.02%)	11.23	16.39	21.18	22.84	9.06	18.01
	Bloom-7B-lora	15.7M (0.22%)	13.81	18.42	22.65	24.68	12.36	19.50
	Bloom-7B-pt	0.8M (0.01%)	12.89	18.60	24.13	27.26	10.61	18.28
	CodeGen-2B-mono-lora	5.2M (0.19%)	13.26	19.71	28.73	32.97	12.62	20.11
	CodeGen-2B-mono-pt	0.5M (0.02%)	14.55	19.71	25.78	31.49	12.16	19.96
	CodeGen-6B-mono-lora	17.3M (0.24%)	14.92	20.99	30.39	33.15	14.45	21.86
	CodeGen-6B-mono-pt	0.1M (0.01%)	**15.10**	**21.18**	**30.76**	**34.62**	**16.83**	**23.51**
	InCoder-1.3B-lora	1.5M (0.12%)	13.63	18.05	24.68	27.62	10.71	18.01
	InCoder-6.7B-lora	8.4M (0.13%)	14.55	20.44	27.26	31.49	13.51	22.10

Table 6: [RQ2] – Effectiveness of the LLMs using ICL. Each cell includes the drop/increase in the metric compared to values reported for the models in bold in Table 5.

Model	EM@1	EM@2	EM@5	EM@10	BLEU	CodeBLEU
PolyCoder-2.7B-icl	4.24 (−3.49 ↓)	6.45 (−4.05 ↓)	8.66 (−6.07 ↓)	9.21 (−6.81 ↓)	5.26 (−4.57 ↓)	12.45 (−5.35 ↓)
Bloom-3B-icl	6.63 (−4.60 ↓)	7.55 (−8.84 ↓)	10.87 (−10.31 ↓)	11.42 (−11.42 ↓)	4.00 (−5.06 ↓)	18.81 (+0.80 ↑)
Bloom-7B-icl	7.00 (−5.89 ↓)	10.50 (−8.10 ↓)	14.00 (−10.13 ↓)	16.02 (−11.24 ↓)	5.27 (−5.34 ↓)	19.55 (+1.27 ↑)
CodeGen-2B-mono-icl	10.31 (−2.95 ↓)	14.36 (−5.35 ↓)	18.97 (−9.76 ↓)	22.28 (−10.69 ↓)	9.86 (−2.76 ↓)	21.04 (+0.93↑)
CodeGen-6B-mono-icl	8.29 (−6.81 ↓)	12.71 (−8.47 ↓)	19.15 (−11.61 ↓)	25.23 (−9.39 ↓)	9.89 (−6.94↓)	21.89 (−1.62 ↓)
InCoder-1.3B-icl	10.66 (−2.97 ↓)	14.73 (−3.32 ↓)	18.78 (−5.90 ↓)	22.28 (−5.34 ↓)	7.66 (−3.05 ↓)	16.94 (−1.07 ↓)
InCoder-6.7B-icl	11.23 (−3.32 ↓)	13.63 (−6.81 ↓)	20.81 (−6.45 ↓)	24.86 (−6.63 ↓)	9.29 (−4.22 ↓)	18.59 (−3.51 ↓)

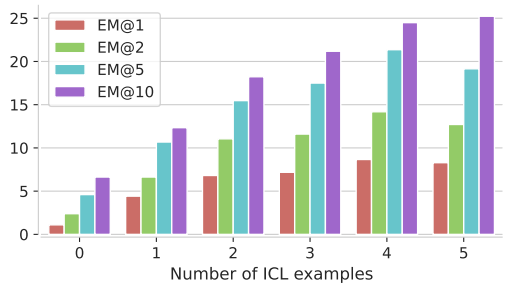

Fig. 3: [RQ2] – Comparison of the effectiveness of CodeGen-6B-mono using ICL with $n = 0..5$.

5.2 RQ2: Comparative Study Between PEFT and ICL Applications on LLMs

In Table 6, we report the performance of the LLMs using ICL with five prompt examples. For each cell, we include the drop or increase in the metric compared to the corresponding best model in Table 5.

Effectiveness of ICL. Firstly, we observe significant decreases in all EM@k metrics for all models when employing ICL. By conducting t-tests across all pairs of models from Table 5 and Table 6, we find statistically significant differences in all EM@k metrics. As the gaps in the metrics are already meaningful, we do not report the p-values to avoid duplicated information. Furthermore, we notice that increasing the number of considered candidates k leads to a larger gap between applications of PEFT and ICL. This outcome suggests that (1) ICL alone is insufficient to adapt an LLM to a new task-specific dataset, and (2) PEFT allows the LLM to generate more contextually meaningful solutions. In other words, PEFT effectively adapts the LLMs to fine-tuning data.

Secondly, we observe that the models exhibiting low EM@k scores, *i.e.*, PolyCoder-2.7B, Bloom-3B, and Bloom-7B, with PEFT also demonstrate the lowest performance with ICL. Under the ICL setting, InCoder-6.7B outperforms CodeGen-6B-mono or shows equivalent performance. Interestingly, CodeGen-2B-mono outperforms CodeGen-6B-mono in EM@1 and EM@2, demonstrating that leveraging lighter models may be a potentially viable option when using ICL.

Impact of ICL examples. To further validate our previous findings, we examine the impact of the number of ICL examples on the effectiveness of CodeGen-6B-mono in Fig. 3. As we increase the number of examples, the model's performance in EM@k metrics improves. That is, by prepending more representative examples to the context, the LLM appears to gain a better understanding of the format of the input format and ground truth data. The rise in

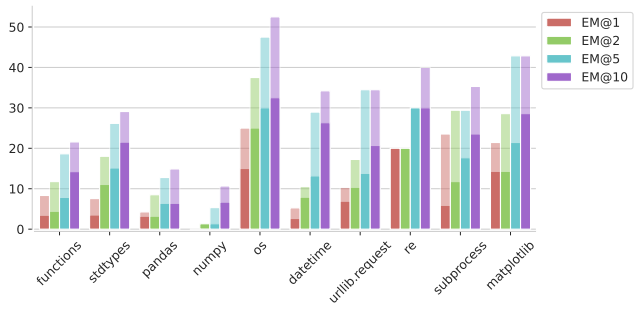

Fig. 4: [RQ3] – Comparison of the effectiveness of CodeGen-6B-mono-pt (*whole bars*) and CodeGen-6B-mono-icl (*dark part of the bars*) for the ten most common modules and third-party libraries in the test dataset.

EM@k stabilizes when reaching $n = 5$, and the model shows higher EM@1, EM@2, and EM@5 when using $n = 4$ ICL examples. Overall, even when considering diverse numbers of prompt examples, ICL does not match up to the performance of PEFT techniques.

Answer to RQ2: LoRA and Prompt tuning are significantly superior to ICL when adapting the LLMs to a new task-specific dataset. ICL is not competitive with PEFT techniques even when increasing the number of prompt examples. For instance, we observe decreasing in each EM@k metric by 29%–45% for CodeGen-6B-mono-icl compared to CodeGen-6B-mono-pt.

5.3 RQ3: Comprehensive Analysis on LLM Performance with PEFT

To analyze the reasons behind the highest performance of PEFT over ICL, we study CodeGen-6B-mono, which is our best LLM that has also shown strong ICL performance in RQ2. We start by comparing the effectiveness of CodeGen-6B-mono-pt and CodeGen-6B-mono-icl on test samples containing the ten most common Python modules and libraries in the test set (see Fig. 2). Then, we report examples in Table 7 to further analyze the variations in the generated codes using both techniques.

Performance of the models. As depicted in Fig. 4, the Prompt tuning application (represented by the whole bars) outperforms the ICL approach (represented by the dark part of the bars) for the CodeGen-6B-mono model in all Python modules and third-party libraries. Specifically, the modules *os*, *subprocess*, and *urllib.request* are where both models exhibit the highest EM@k scores. Overall, fine-tuning with Prompt tuning enables significantly improving the effectiveness of the model for all modules and libraries. This result further confirms that fine-tuning is crucial for better adaptation of the LLM to the task-specific data, and underscores the relevance of leveraging PEFT techniques in this context.

Nonetheless, we observe poor performance on *pandas* and *numpy* libraries with both methods. Prompt tuning still improves the EM@k for both libraries, but the EM@k remain low, although the training set contains 236 and 93 samples involving *pandas* and *numpy*, respectively. We acknowledge that both libraries may offer multiple functionally correct solutions for a given problem, and we recognize that the EM metric may not fully capture the functional correctness. We have conducted an analysis of the generated code for both methods on specific test samples and have identified numerous cases where the models produce alternative solutions that are functionally correct. We report our findings across three examples in Table 7.

Case study. In the first example (*pandas*), both models provide functionally correct solutions. The solution generated by CodeGen-6B-mono-pt is preferred as it matches the ground truth and constitutes a more concise and readable solution. It is noteworthy that both models demonstrate an understanding of the NL intent, as evident from the presence of hints in bold within their solutions.

In the second example (*numpy*), both models generate functionally correct solutions without matching the ground truth. Interestingly, CodeGen-6B-mono-pt generates a more efficient one that modifies the array in place, whereas the other solution involves the creation of a new array. We note that the ground truth involves reshaping the array 'A', which is not specified in the NL intent. The lack of informative NL intent prevents both models from generating the exact solution, emphasizing the importance of providing comprehensive context in the input prompt.

Finally, in the third example (*os*), both models provide different implementations of the NL intent. CodeGen-6B-mono-pt generates the exact solution representative of the NL intent, whereas CodeGen-6B-mono-icl deviates by providing a solution specifically for reading CSV data from stdin. This discrepancy in alignment with the NL intent may be attributed to the influence of ICL examples, especially one involving the *pandas* library.

Note that due to page limit, we only discuss these three examples. We include more examples in our replication package to showcase that our following conclusion applies to a wide range of test samples.

Answer to RQ3: Based on the studied examples, fine-tuning the LLMs with PEFT techniques improves the alignment of the candidate solutions provided by the LLMs with the natural language intents. Furthermore, both models, *i.e.*, with PEFT and ICL, generate alternative correct solutions that are not captured by the EM@k metrics.

6 THREATS TO VALIDITY

External validity. We identified a few threats to the external validity of our experiments. One main threat relates to the choice of our small language models and LLMs. We mitigated this threat by carefully selecting a diverse set of models, as explained in Section 4.3. These models encompass various families of LLMs, trained on distinct pre-training data and learning objectives, and varying in size. Additionally, we selected robust small language models, such as CodeGen-350M-mono, to thoroughly evaluate full fine-tuning against the PEFT of LLMs.

Another external threat to the validity is related to the quality of the fine-tuning dataset. To alleviate this concern, we chose the CoNaLa dataset, which contains high-quality examples mined from StackOverflow posts. Additionally, this dataset has been representatively used by multiple prior studies [45, 67, 87] on code generation

Table 7: [RQ3] – Examples for pandas and numpy libraries and Python's os module.

NL intent: converting dictionary '**d**' into a dataframe '**pd**' with keys as data for column '**Date**' and the corresponding values as data for column '**DateValue**'	
Ground truth:	`pd.DataFrame(list(d.items()), columns=['Date', 'DateValue'])`
CodeGen-6B-mono-pt:	`pd.DataFrame(list(d.items()), columns=['Date', 'DateValue'])`
CodeGen-6B-mono-icl:	`pd.DataFrame([(k, v) for k, v in d.items()], columns=['Date', 'DateValue'])`
NL intent: replace all elements in array '**A**' that are not present in array '**[1, 3, 4]**' with zeros	
Ground truth:	`np.where(np.in1d(A, [1, 3, 4]).reshape(A.shape), A, 0)`
CodeGen-6B-mono-pt:	`A[~np.in1d(A, [1, 3, 4])] = 0`
CodeGen-6B-mono-icl:	`np.where(np.in1d(A, [1, 3, 4]), A, 0)`
NL intent: read a file from redirected stdin and save to variable '**result**'	
Ground truth:	`result = sys.stdin.read()`
CodeGen-6B-mono-pt:	`result = sys.stdin.read()`
CodeGen-6B-mono-icl:	`result = pd.read_csv(sys.stdin)`

task. Furthermore, the authors enriched each natural language intent with hints, enhancing the alignment of input prompts with possible human intents. Another threat relates to the monolingual aspect of the CoNaLa dataset. We studied full fine-tuning, PEFT, and ICL for code generation of Python code snippets. However, we anticipate that PEFT techniques is also applicable to other programming languages, considering the impressive generation capabilities of LLMs on a diverse range of programming languages [3, 8].

The representativeness of the selected prompt examples for ICL poses an additional external threat to the validity of our experiments. To mitigate this threat, we carefully chose prompt examples covering different Python modules and third-party libraries. Although selecting different ICL examples could introduce variation in the evaluation metrics [17, 64], we believe that our results sufficiently cover the initial trial for the field of software engineering.

Finally, the limited size of examples in RQ3 is another threat to external validity. To alleviate this threat, we selected examples covering different libraries and modules. To further mitigate this threat, we explore more examples in our replication package where the models generate alternative correct solutions.

Internal validity. The hyperparameter choices for the PEFT techniques and ICL constitute the main threat to internal validity. For both PEFT techniques, *i.e.*, LoRA and Prompt tuning, we fine-tuned each LLM using various hyperparameter configurations. We specifically tuned the hyperparameters which have the greatest impact on the model performance according to the original papers of each technique (see Section 4.6). As for ICL, we evaluated each LLM with a different number of prompt examples and reported the evaluation metrics of the best model for a fair comparison with PEFT approaches.

Construct validity. We identified one main threat to construct validity related to the evaluation metrics. To mitigate this threat, we selected evaluation metrics widely used in prior works [30, 39, 66, 77] on code generation. Furthermore, we evaluate each approach using EM@k, which enriched our analysis by computing the exact match over different ranges of code candidates. Finally, we did not use Pass@k metrics as the CoNaLa dataset does not come with unit tests.

7 DISCUSSION

Our study on the exploration of PEFT techniques for LLMs of code demonstrates the positive impact of these applications to efficiently adapt LLMs of code to task-specific datasets. We are confident that PEFT techniques can significantly impact the domain of deep learning for code, with a wide array of potential applications. Our findings highlight two potential areas of future work: improving code generation datasets and further exploring the applicability of PEFT for models of code.

Code generation datasets. The quality of the dataset to fine-tune and evaluate LLMs for code generation is crucial. In this study, we leveraged the CoNaLa dataset, which includes manually annotated hints in the natural language intent. In our comprehensive analysis of the performance of CodeGen-6B-mono (see Section 5.3), we demonstrate that hints are always present in the generated solution. These hints thus provide meaningful context to the LLM and make it more likely to generate the correct solution. However, in our second example in Section 5.3, we also highlight the important missing context in the natural language intent which prevents the model to generate an adequate solution. This observation demonstrates the importance of designing good datasets (*i.e.*, with good input prompts) to enhance the capability of the LLMs for code generation.

PEFT for models of code. In this study, we conducted all our experiments with limited computational resources. As a result, the adoption of the full fine-tuning approach for LLMs was intractable and unfeasible. Previous studies [7, 13, 14] have highlighted that PEFT techniques can often achieve comparable performance to full fine-tuning. Moreover, PEFT techniques offer increased robustness, and, in some cases, superior performance over full fine-tuning [13, 14, 64]. Therefore, conducting a comprehensive comparison between PEFT techniques and full fine-tuning of LLMs would provide valuable insights into whether similar benefits can be observed in LLMs of code. Note that we did not study each technique in terms of time cost. While LoRA and Prompt tuning require additional fine-tuning time compared to ICL, both techniques do not yield any additional time cost at inference compared to ICL, making them as usable as ICL in practice.

Our work focuses on Python code generation. To maintain a coherent and manageable scope for our study, we refrained from incorporating supplementary tasks and datasets, as it could lead

to an overly extensive range of orthogonal analyses. Instead, we focused on code generation, a key task in software engineering, employing a high-quality dataset. We anticipate that exploring PEFT techniques for LLMs across diverse datasets and tasks holds great promise as a direction for future research.

From a different angle, prior studies [18, 71, 76] highlighted the need to consider pre-trained language models and LLMs of code in continual learning settings. In this paradigm, the model needs to adapt to new data over time while preserving good performance on previously seen data. In the specific setting of continuously adapting LLMs over time, PEFT techniques potentially offer valuable benefits. Nonetheless, it is yet to determine whether PEFT techniques are able to efficiently adapt LLMs under a continual learning setting for code-related tasks, without forgetting about past knowledge.

8 RELATED WORK

8.1 Automated Code Generation

A significant portion of code generation techniques [2, 5, 20, 58] relies on deep-learning-based approaches. For instance, the earliest approach [36] utilized sequence-to-sequence models to facilitate the transformation from natural language to code snippets. Mukherjee et al. [42] introduced a generative modeling approach for source code that incorporates the utilization of a static analysis tool, enabling the model to produce code that aligns with the patterns and characteristics in the existing codebase. Further research works [30, 77] proposed the advancement of pre-trained models for code such as CodeBERT [15] and CodeT5 [66] on automated code generation.

The latest trend in automated code generation revolves around leveraging LLMs like GPT models [46] due to their remarkable breakthroughs in this domain. One notable example is Codex, developed by Chen et al. [10], which is a fine-tuned version of GPT-3. Another noteworthy model following the success of Codex is CodeGen [44]. This approach adopts the concept of a multi-turn programming benchmark, effectively democratizing the breakthrough performance achieved by Codex and bringing it to a broader audience.

Indeed, these LLMs have demonstrated success in various software engineering downstream tasks, including automated code generation. However, fine-tuning these models to achieve optimal performance necessitates a substantial amount of task-specific data. Acquiring such data can be costly and time-consuming, presenting a challenge for researchers and developers looking to harness the full potential of LLMs in specific domains like code generation. Furthermore, in traditional fine-tuning processes, all the parameters of the model need to be updated, which can result in significant computational expenses as well. We believe that our study can shed light on more efficient and cost-effective approaches to fine-tuning these LLMs, mitigating the data and computational burdens associated with their adoption.

8.2 Efficient Adaptation of Models of Code

Efficient adaptation of models of code involves the utilization of techniques to efficiently adapt a model to a task-specific dataset (see Section 2). In this context, the term "efficient" refers to updating a low number of parameters in a fine-tuning setting, *e.g*, using LoRA, or utilizing parameter-free techniques such as prompting and ICL.

Most prior research has concentrated on employing ICL and prompting to adapt models to diverse code-related tasks. Gao et al. [17] showcased the advantages of ICL in tasks like bug fixing, code summarization, and program synthesis. They highlighted that the model's performance on downstream tasks is influenced by multiple factors, including the selection, quantity, and order of prompt examples. Other studies [48, 73] also demonstrated that pre-trained language models and LLMs like Codex can effectively handle bug fixing and automated program repair using ICL. Moreover, Geng et al. [19] demonstrated the capability of Codex to generate multi-intent comment generation to describe the functionality of a method or its implementation details, for instance. The selection of relevant prompts for a task with ICL is crucial to ensure the good performance of a LLM. Prior works [43, 87] designed selection techniques to retrieve highly relevant prompt examples tailored to downstream tasks, outperforming random selection methods. Lastly, recent research [55] highlighted the advantages of retrieving prompt examples at the repository-level, providing LLMs with valuable contextual information in the prompts. In this study, we leveraged ICL without the intention of fully exploring its potential (see Section 6). Instead, we opted for a simple implementation of ICL by carefully selecting the prompt examples. Extending this study to include more of the aforementioned ICL approaches could further enrich the comparison between ICL and PEFT techniques for code.

Regarding PEFT techniques, prior research in code intelligence has focused on Prompt tuning [29], Prefix-tuning [31] and Adapters [22, 24]). Wang et al. [64] initiated the usage of Prompt tuning for code-related tasks and demonstrated its superiority over full fine-tuning of CodeT5 and CodeBERT in defect prediction, code summarization, and code translation. Similarly, Choi et al. [11] designed a code-specific Prefix-tuning approach within a sequence-to-sequence architecture for generation tasks. Our study differs from these two previous works as they focus on small language models, whereas we propose the first comprehensive study of PEFT techniques with LLMs for code generation. Moreover, our study includes LoRA, which none of the previous work in code intelligence consider for efficiently tuning LLMs of code. Finally, Wang et al. [65] showcased the superiority of utilizing Adapters for fine-tuning pre-trained language models over full fine-tuning. Our research diverges from this prior work, as we concentrate on state-of-the-art LLMs. Although we did not incorporate Adapters in our investigation, we believe that LoRA and Prompt tuning provide a sufficiently thorough analysis of PEFT techniques. Nevertheless, we acknowledge the potential value of exploring more PEFT techniques for diverse code intelligence tasks in the future.

9 CONCLUSION AND FUTURE WORK

In this study, we have demonstrated the benefits of leveraging PEFT techniques for LLMs of code. Our comparative study explored the performance of LoRA and Prompt tuning, with a wide range of models covering four families of LLMs. The results show the superiority of LLMs fine-tuned with both PEFT techniques over fully fine-tuned smalr language models and LLMs with ICL for code generation.

Furthermore, we illustrated the relevance of PEFT techniques under a limited resources scenario, effectively mitigating the computational burden and impractical aspect of full fine-tuning. Our study consists of the first comprehensive study on the utilization of PEFT techniques for LLMs in software engineering. We believe our work could initiate a new research direction on the utilization of PEFT techniques in software engineering and are excited by the potential broader impact of these techniques. In future work, we intend to study PEFT techniques for other tasks such as code search or code comment generation. Finally, we aim at validating further reliance on PEFT techniques under continual learning settings for code.

REFERENCES

[1] Toufique Ahmed and Premkumar Devanbu. 2022. Few-shot training LLMs for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[2] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural language models of code. In *International conference on machine learning*. PMLR, 245–256.

[3] Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, et al. 2022. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868* (2022).

[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[5] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989* (2016).

[6] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. *GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow.* https://doi.org/10.5281/zenodo.5297715 If you use this software, please cite it using these metadata..

[7] 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.

[8] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2023. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering* (2023).

[9] Christel Chappuis, Valérie Zermatten, Sylvain Lobry, Bertrand Le Saux, and Devis Tuia. 2022. Prompt-RSVQA: Prompting visual context to a language model for remote sensing visual question answering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 1372–1381.

[10] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[11] YunSeok Choi and Jee-Hyong Lee. 2023. CodePrompt: Task-Agnostic Prefix Tuning for Program and Language Generation. In *Findings of the Association for Computational Linguistics: ACL 2023*. 5282–5297.

[12] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311* (2022).

[13] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. 2022. Delta tuning: A comprehensive study of parameter efficient methods for pre-trained language models. *arXiv preprint arXiv:2203.06904* (2022).

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

[15] 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).

[16] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations*.

[17] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, and Michael R Lyu. 2023. Constructing Effective In-Context Demonstration for Code Intelligence Tasks: An Empirical Study. *arXiv preprint arXiv:2304.07575* (2023).

[18] Shuzheng Gao, Hongyu Zhang, Cuiyun Gao, and Chaozheng Wang. 2023. Keeping Pace with Ever-Increasing Data: Towards Continual Learning of Code Intelligence Models. *arXiv preprint arXiv:2302.03482* (2023).

[19] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning. (2024).

[20] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. *arXiv preprint arXiv:1808.10025* (2018).

[21] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The Curious Case of Neural Text Degeneration. arXiv:1904.09751 [cs.CL]

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

[23] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).

[24] Zhiqiang Hu, Yihuai Lan, Lei Wang, Wanyu Xu, Ee-Peng Lim, Roy Ka-Wei Lee, Lidong Bing, and Soujanya Poria. 2023. LLM-Adapters: An Adapter Family for Parameter-Efficient Fine-Tuning of Large Language Models. *arXiv preprint arXiv:2304.01933* (2023).

[25] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*. 1219–1231.

[26] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5131–5140.

[27] Enkelejda Kasneci, Kathrin Seßler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, et al. 2023. ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and Individual Differences* 103 (2023), 102274.

[28] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems* 35 (2022), 22199–22213.

[29] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The Power of Scale for Parameter-Efficient Prompt Tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 3045–3059.

[30] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. Skcoder: A sketch-based approach for automatic code generation. *arXiv preprint arXiv:2302.06144* (2023).

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

[32] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[33] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. 2022. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110* (2022).

[34] W Liang, M Yuksekgonul, Y Mao, E Wu, and J Zou. 2023. GPT detectors are biased against non-native English writers (arXiv: 2304.02819). arXiv.

[35] Xi Victoria Lin, Todor Mihaylov, Mikel Artetxe, Tianlu Wang, Shuohui Chen, Daniel Simig, Myle Ott, Naman Goyal, Shruti Bhosale, Jingfei Du, Ramakanth Pasunuru, Sam Shleifer, Punit Singh Koura, Vishrav Chaudhary, Brian O'Horo, Jeff Wang, Luke Zettlemoyer, Zornitsa Kozareva, Mona Diab, Veselin Stoyanov, and Xian Li. 2022. Few-shot Learning with Multilingual Language Models. arXiv:2112.10668 [cs.CL]

[36] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744* (2016).

[37] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. 2022. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems* 35 (2022), 1950–1965.

[38] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.

[39] 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).

[40] Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heintz, and Dan Roth. 2021. Recent advances in natural language processing via large pre-trained language models: A survey. *Comput. Surveys* (2021).

[41] Sewon Min, Mike Lewis, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2021. Metaicl: Learning to learn in context. *arXiv preprint arXiv:2110.15943* (2021).

[42] Rohan Mukherjee, Yeming Wen, Dipak Chaudhari, Thomas Reps, Swarat Chaudhuri, and Christopher Jermaine. 2021. Neural program generation modulo static analysis. *Advances in Neural Information Processing Systems* 34 (2021), 18984–18996.

[43] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*.

[44] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. arXiv:2203.13474 [cs.LG]

[45] Sajad Norouzi, Keyi Tang, and Yanshuai Cao. 2021. Code generation from natural language with less prior knowledge and more monolingual data. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*. 776–785.

[46] R OpenAI. 2023. GPT-4 technical report. *arXiv* (2023), 2303–08774.

[47] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems* 35 (2022), 27730–27744.

[48] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's codex fix bugs? an evaluation on QuixBugs. In *Proceedings of the Third International Workshop on Automated Program Repair*. 69–75.

[49] 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.

[50] 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).

[51] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. 27–43.

[52] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2022. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100* (2022).

[53] Zhenwei Shao, Zhou Yu, Meng Wang, and Jun Yu. 2023. Prompting large language models with answer heuristics for knowledge-based visual question answering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 14974–14983.

[54] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL]

[55] Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*. PMLR, 31693–31715.

[56] Karan Singhal, Shekoofeh Azizi, Tao Tu, S Sara Mahdavi, Jason Wei, Hyung Won Chung, Nathan Scales, Ajay Tanwani, Heather Cole-Lewis, Stephen Pfohl, et al. 2023. Large language models encode clinical knowledge. *Nature* (2023), 1–9.

[57] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and policy considerations for deep learning in NLP. *arXiv preprint arXiv:1906.02243* (2019).

[58] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8984–8991.

[59] Alex Tamkin, Miles Brundage, Jack Clark, and Deep Ganguli. 2021. Understanding the capabilities, limitations, and societal impact of large language models. *arXiv preprint arXiv:2102.02503* (2021).

[60] Marcos Treviso, Ji-Ung Lee, Tianchu Ji, Betty van Aken, Qingqing Cao, Manuel R Ciosici, Michael Hassid, Kenneth Heafield, Sara Hooker, Colin Raffel, et al. 2023. Efficient methods for natural language processing: A survey. *Transactions of the Association for Computational Linguistics* 11 (2023), 826–860.

[61] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*. 1–7.

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

[63] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax.

[64] 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*. 382–394.

[65] Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, and Xiangke Liao. 2023. One Adapter for All Programming Languages? Adapter Tuning for Code Search and Summarization. *arXiv preprint arXiv:2303.15822* (2023).

[66] 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).

[67] Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. 2022. Execution-Based Evaluation for Open-Domain Code Generation. *arXiv preprint arXiv:2212.10481* (2022).

[68] Albert Webson and Ellie Pavlick. 2021. Do prompt-based models really understand the meaning of their prompts? *arXiv preprint arXiv:2109.01247* (2021).

[69] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652* (2021).

[70] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. 2022. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682* (2022).

[71] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. 2023. On the Usage of Continual Learning for Out-of-Distribution Generalization in Pre-trained Language Models of Code. *arXiv preprint arXiv:2305.04106* (2023).

[72] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771* (2019).

[73] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*.

[74] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 959–971.

[75] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code *(MAPS 2022)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3520312.3534862

[76] Prateek Yadav, Qing Sun, Hantian Ding, Xiaopeng Li, Dejiao Zhang, Ming Tan, Xiaofei Ma, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, et al. 2023. Exploring Continual Learning for Code Generation Models. *arXiv preprint arXiv:2307.02435* (2023).

[77] Guang Yang, Yu Zhou, Xiang Chen, Xiangyu Zhang, Tingting Han, and Taolue Chen. 2023. ExploitGen: Template-augmented exploit code generation based on CodeBERT. *Journal of Systems and Software* 197 (2023), 111577.

[78] Yue Yang, Artemis Panagopoulou, Shenghao Zhou, Daniel Jin, Chris Callison-Burch, and Mark Yatskar. 2023. Language in a bottle: Language model guided concept bottlenecks for interpretable image classification. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 19187–19197.

[79] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 476–486. https://doi.org/10.1145/3196398.3196408

[80] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th international conference on mining software repositories (MSR)*. IEEE, 476–486.

[81] Bowen Zhang, Xianghua Fu, Daijun Ding, Hu Huang, Yangyang Li, and Liwen Jing. 2023. Investigating Chain-of-thought with ChatGPT for Stance Detection on Social Media. *arXiv preprint arXiv:2304.03087* (2023).

[82] Pengchuan Zhang, Xiujun Li, Xiaowei Hu, Jianwei Yang, Lei Zhang, Lijuan Wang, Yejin Choi, and Jianfeng Gao. 2021. Vinvl: Revisiting visual representations in vision-language models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 5579–5588.

[83] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068* (2022).

[84] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey

of large language models. *arXiv preprint arXiv:2303.18223* (2023).

[85] Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. 2021. Calibrate before use: Improving few-shot performance of language models. In *International Conference on Machine Learning*. PMLR, 12697–12706.

[86] Kaiyang Zhou, Jingkang Yang, Chen Change Loy, and Ziwei Liu. 2022. Learning to prompt for vision-language models. *International Journal of Computer Vision* 130, 9 (2022), 2337–2348.

[87] Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao Jiang, and Graham Neubig. 2023. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*.