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

MARTIN WEYSSOW, DIRO, University of Montreal, Canada
XIN ZHOU*, Singapore Management University, Singapore
KISUB KIM, Singapore Management University, Singapore
DAVID LO, Singapore Management University, Singapore
HOUARI SAHRAOUI, DIRO, University of Montreal, Canada

Large language models (LLMs) demonstrate impressive capabilities to generate accurate code snippets given natural language intents in a zero-shot manner, *i.e.*, without the need for specific fine-tuning. While prior studies have highlighted the advantages of fine-tuning LLMs, this process incurs high computational costs, making it impractical in resource-scarce environments, particularly for models with billions of parameters. To address these challenges, previous research explored in-context learning (ICL) and retrieval-augmented generation (RAG) as strategies to guide the LLM generative process with task-specific prompt examples. However, ICL and RAG introduce inconveniences, such as the need for designing contextually relevant prompts and the absence of learning task-specific parameters, thereby limiting downstream task performance. In this context, we foresee parameter-efficient fine-tuning (PEFT) as a promising approach to efficiently specialize LLMs to task-specific data while maintaining reasonable resource consumption. In this paper, we deliver a comprehensive study of PEFT techniques for LLMs in the context of automated code generation. Our comprehensive investigation of PEFT techniques for LLMs reveals their superiority and potential over ICL and RAG across a diverse set of LLMs and three representative Python code generation datasets: Conala, CodeAlpacaPy, and APPS. Furthermore, our study highlights the potential for tuning larger LLMs and significant reductions in memory usage by combining PEFT with quantization. Therefore, this study opens opportunities for broader applications of PEFT in software engineering scenarios.

CCS Concepts: • **Software and its engineering → Software creation and management**; **Software development techniques**; • **Computing methodologies → Natural language processing**; **Machine learning**;

Additional Key Words and Phrases: code generation, large language models, parameter-efficient fine-tuning, quantization, retrieval-augmented generation, empirical study

1 INTRODUCTION

Large Language Models (LLMs) based on the Transformer architecture [67], demonstrate significant potential in diverse domains, including natural language processing (NLP) [29, 44, 76], computer vision [7, 59, 85], and software engineering [9, 66, 82]. These models excel in generating high-quality content given natural language intents in

*Corresponding author.

Authors' addresses: Martin Weyssow, martin.weyssow@umontreal.ca, DIRO, University of Montreal, Canada; Xin Zhou, xinzhou.2020@phdcs.smu.edu.sg, Singapore Management University, Singapore; Kisub Kim, falconlk00@gmail.com, Singapore Management University, Singapore; David Lo, davidlo@smu.edu.sg, Singapore Management University, Singapore; Houari Sahraoui, sahraouh@iro.umontreal.ca, DIRO, University of Montreal, Canada.

ACM 1557-7392/2025/1-ART
https://doi.org/10.1145/3714461

zero-shot, *i.e.*, without fine-tuning. This capability has sparked considerable interest in the software engineering field for automating code-related tasks such as program repair [27, 80, 81] and code generation [3, 9, 48].

While the zero-shot capabilities of LLMs are impressive, their full potential often emerges through fine-tuning [54, 75]. Specifically, fine-tuning an LLM to task-specific data allows it to learn and encode knowledge of the potentially highly contextual data at hand and thus generate more meaningful 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 [62]. To mitigate this computational burden, prior studies in software engineering [53, 80, 91] have investigated prompt-engineering techniques such as In-Context Learning (ICL) [5, 54] and Retrieval-Augmented Generation (RAG) [31]. ICL consists of providing prompt examples of the task to the LLM, guiding it to generate contextually appropriate content without any fine-tuning involved. These examples can be manually created or randomly selected from a relevant training dataset. This technique has already shown promising results for code-related tasks, including automated program repair [80], bug fixing [53], and code generation [61, 73, 91]. Expanding upon ICL, RAG offers a more robust and powerful alternative that incorporate a knowledge retrieval system at inference. Using RAG, a retrieval model fetches relevant information from an indexed corpus, such as code documentation or similar code snippets to the input problem. The retrieved information is then added to the input prompt to guide generation. Unlike ICL, which relies on pre-selected examples that may not always be tailored to the specific input, RAG dynamically adapts to each individual input problem, providing more relevant context. This technique has demonstrated significant improvements in software engineering tasks such as code generation and summarization [39, 52, 91], code completion [41], and program repair [70].

Although ICL and RAG provide a viable alternative to full fine-tuning, it operates at inference time and does not involve learning task-specific parameters, which may prevent the LLM from capturing fine-grained information about the task and result in a 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 [10, 57, 68, 69] 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 (SLMs) (<0.25B parameters) such as CodeBERT [16] and CodeT5 [72] and overlooked the applicability of PEFT techniques to LLMs (≥1B parameters), leaving an important research gap. Given the growing ubiquity of LLMs, we believe addressing this gap is paramount 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 broaden the accessibility and utility of LLMs in scarce computation settings and alleviate the dependence of practitioners on large computational infrastructures.

In this paper, we present an empirical study on the usage of existing PEFT techniques with 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 [9, 48, 50]. Our objective is twofold. First, we aim to assess the code generation capabilities of LLMs using existing PEFT techniques such as LoRA [24] and QLoRA [13] on datasets without test cases, including Conala [91] and CodeAlpacaPy [8], as well as the APPS dataset [22] with test cases. Second, we seek to compare the effectiveness of LLMs tuned with these PEFT techniques against SLMs, ICL, and RAG. 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 four research questions that guide our study:

– RQ1: How do LLMs and SLMs perform using ICL on the Conala and CodeAlpacaPy datasets?

– RQ2: How do LLMs and SLMs perform using PEFT techniques on the Conala and CodeAlpacaPy datasets?
– RQ3: How does LoRA compare with ICL and RAG on the Conala and CodeAlpacaPy datasets?
– RQ4: Can we enhance the effectiveness of LLMs for code generation in the APPS dataset using LoRA and QLoRA?

Altogether, answering these four research questions fulfills both objectives of this empirical study. Our first three RQs focus on evaluating SLMs and LLMs for code generation on the Conala and CodeAlpaca datasets. In RQ1, we illustrate the baseline effectiveness of SLMs and LLMs using ICL, which retrieves random examples from the training set to guide the model in generating code. By addressing RQ2, we gain a comprehensive understanding of how effective SLMs and LLMs are when using different PEFT techniques. In RQ3, we conduct a comparative study of the effectiveness of LoRA with ICL and RAG, a strong baseline that dynamically retrieves relevant examples by selecting those closest to the test instructions from the training set. Finally, to showcase the potential broader impact of PEFT, we study in RQ4 whether tuning LLMs using LoRA and QLoRA can improve their effectiveness on APPS, a challenging benchmark with test cases.

To address these RQs, we conduct experiments on three datasets, APPS [22], Conala [86], and CodeAlpacaPy specifically curated from CodeAlpaca [8] for Python code generation. Conversely to evaluation datasets such as HumanEval [9], the APPS, Conala and CodeAlpaca datasets, widely used in prior code generation studies [49, 71, 73, 73, 88, 91], include sufficient training examples that can be employed for fine-tuning. For a comprehensive comparative analysis, we select four distinct model families: CodeT5+ [71], CodeGen [48], CodeGen2 [47], and CodeLlama [56], including eight 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, our study incorporates six PEFT techniques: LoRA [24], IA3 [37], Prompt tuning [30], and Prefix tuning [33]. In addition, we explore QLoRA [13] with 8-bit and 4-bit quantization, which combines LoRA and model quantization. Unlike ICL and RAG, these techniques entail learning new parameters to tune the LLMs for the specific downstream task. Our main findings are the following:

– ICL drastically improves the effectiveness of all models compared to a zero-shot prompt for code generation on Conala and CodeAlpacaPy.
– Increasing the number of ICL examples does not always lead to improvement in effectiveness. Models achieve peak effectiveness with eight and four examples for Conala and CodeAlpacaPy, respectively.
– LLMs fine-tuned with LoRA, IA3, and Prompt tuning, *i.e.,* a few millions of parameters, consistently outperform SLMs fully fine-tuned with hundreds of millions of parameters.
– Among PEFT techniques, LoRA achieves the highest effectiveness for the LLMs and SLMs.
– QLoRA considerably reduces memory usage, achieving up to a 2-fold decrease compared to LoRA while improving or preserving the models' effectiveness. Furthermore, QLoRA enables the fine-tuning of LLMs up to 34B parameters for less than 24GB of GPU memory.
– LoRA significantly enhances the performance of all models compared to ICL and RAG for code generation on Conala and CodeAlpacaPy.
– LoRA and QLoRA improve CodeLlama-7B-Instruct's effectiveness for code generation on the APPS dataset.

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 following:

– We conduct a comprehensive empirical study of six PEFT techniques, *i.e.*, LoRA, IA3, Prompt tuning, Prefix tuning, QLoRA-8bit, and QLoRA-4bit, for Python code generation over a broad range of SLMs and LLMs.
– A comprehensive comparison and analysis of PEFT techniques against ICL and RAG for LLMs on code generation.

– We demonstrate the practicality of leveraging PEFT techniques to effectively fine-tune LLMs of code and reduce the computational burden associated with full fine-tuning, showcasing their potential broader applications in software engineering.

2 BACKGROUND

2.1 In-Context Learning (ICL) and Retrieval-Augmented Generation (RAG)

As one of the specific types of LLM-related techniques, ICL has emerged as an effective technique [5, 11, 35, 45, 51]. ICL seeks to improve the abilities of LLMs by integrating context-specific information, in the form of an input prompt or instruction template, during the inference and thus without the need to perform gradient-based training. Therefore, by considering the context, the model becomes more capable of generating 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 [37, 74, 90].

RAG is a more sophisticated approach to inject examples into input prompts at inference. Unlike ICL that select random examples, RAG relies on a retrieval model that dynamically retrieves examples from a dataset that are close to a query. In practice, the query can be formulated using information from the test example at test time, such as the coding problem for the case of code generation. Altogether, RAG allows injection more relevant information in the input prompt than ICL and has been succesfully applied to software engineering tasks, such as code generation [52, 91], code summarization [39, 52], and code completion [41]. Nonetheless, both ICL and RAG suffer from a few limitations. One concerns the introduction of extra input tokens in the prompt, which may be infeasible when the contextual information is too large. Another limitation is the reliance on the quality and relevance of the retrieved examples. In RAG, the retrieval model must accurately find examples that are genuinely similar or useful for the test query. If the retrieval mechanism fails to identify appropriate examples, it can inject irrelevant or misleading information into the prompt, ultimately degrading performance.

2.2 Parameter-Efficient Fine-Tuning (PEFT)

PEFT refer to the utilization of techniques that optimize the fine-tuning process of LLMs by selectively updating a subset of parameters instead of updating the entire model's parameters [14]. Technically, PEFT techniques focus on learning a small number of parameters for the task at hand by designing additional layers [23], adding prepending additional tokens [30, 33], decomposing weight gradients into specific matrices [24]. One of the representative cutting-edge PEFT techniques is LOw-Rank Adaptation of LLMs (LoRA) [24]. The technique consists of freezing the model weights and injecting low-rank trainable matrices into the attention layers of the Transformer architecture [67], thereby drastically reducing the number of trainable parameters. We employ LoRA as one of our PEFT technique since it has been widely used in NLP [14, 37, 65] and showed promising performance. We also employ IA3 which intends to improve upon LoRA and further reduces the amount of trainable parameters [37]. In addition to LoRA and IA3, we also include Prompt tuning [30] and Prefix tuning [30] in our study. Prompt tuning involves the process of prepending virtual tokens to the input tokens of the LLM, whereas Prefix tuning inserts virtual tokens in all the layers of the target model and thus requires learning more parameters. These virtual tokens are differentiable, allowing them to be learned through backpropagation during fine-tuning, while the rest of the LLM remains frozen. Furthermore, QLoRA [13] combines LoRA with model quantization, enabling the fine-tuning of LLMs with less GPU memory by reducing the precision of floating point data types within the model.

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

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

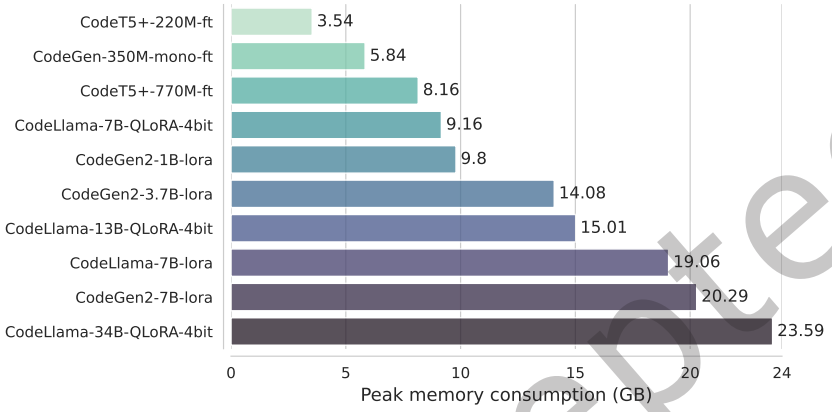


Fig. 1. Peak GPU memory consumption during models fine-tuning using full fine-tuning (ft), LoRA, and QLoRA.

3 APPLYING 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.

For instance, a software engineer with access to only a single consumer GPU (e.g., 24GB of VRAM) may find full fine-tuning impractical due to the significant memory demands. The rapid increase in model size and the number of trainable parameters exacerbates this issue. Despite its effectiveness, full fine-tuning comes at a steep computational cost [3, 15, 51]., underscoring a computation-effectiveness trade-off (see Table 1).

To address these limitations, alternative approaches like ICL and RAG have gained attention. ICL and RAG offer a low-computation option by eliminating the need for parameter updates. However, these techniques comes with its own set of challenges, including the selection of representative examples and sensitivity to prompt design [37, 74, 90]. In practice, this can result in lower effectiveness compared to fine-tuning, particularly for highly contextual tasks prevalent in software engineering.

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. PEFT methods, such as LoRA and QLoRA, limit the number of parameters being updated, thus reducing memory consumption while maintaining effectiveness competitive with full fine-tuning. This makes PEFT particularly well-suited for practitioners with limited access to computational resources. As illustrated in Table 1, PEFT strike an optimal balance between computational cost and effectiveness. Furthermore, Fig. 1 shows that by employing PEFT techniques like LoRA, practitioners can fine-tune models such as CodeLlama-7B without exceeding 19GB of GPU memory. For even larger models, such as CodeLlama-34B, QLoRA with quantization enables fine-tuning within the constraints of a 24GB VRAM GPU.

In conclusion, PEFT empower software engineers to overcome resource limitations, allowing for effective LLM fine-tuning in highly contextual tasks without relying on expensive computational infrastructures. This makes PEFT not only a practical but also an essential tool for democratizing access to LLM capabilities.

4 METHODOLOGY

In this section, we present 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 GPU. The main objective of our study is to demonstrate whether the fine-tuning of LLMs through PEFT 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 and SLMs perform using ICL on the Conala and CodeAlpacaPy datasets?** We study the baseline effectiveness of LLMs ($\geq$ 1B parameters) and SLMs ($<$ 1B parameters) for code generation using the zero-shot prompt and ICL, where n randomly selected examples are added to the input prompt. We test each model with up to 16 ICL examples, due to our limited computation resources.
  We study the effectiveness of a large spectrum of SLMs and LLMs for code generation on two datasets covering codes of various lengths. 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: **How do LLMs and SLMs perform using PEFT techniques on the Conala and CodeAlpacaPy datasets?** In this RQ, we investigate whether PEFT techniques consistently outperform ICL for SLMs and LLMs. We compare the best-performing configurations of ICL in RQ1 with PEFT techniques, including LoRA, IA3, Prompt tuning, Prefix tuning. Furthermore, we also investigate the effect of quantization with QLoRA-8bit and QLoRA-4bit on our best-performing model and larger variants.
  For SLMs, we also include a comparison with full-parameter fine-tuning, as commonly used in previous SE studies [16, 72, 77, 92]. We do not include full-parameter fine-tuning for LLMs, as it is not feasible within our computational budget.
- RQ3: **How does LoRA compare with ICL and RAG on the Conala and CodeAlpacaPy datasets?** In this RQ, we compare the effectiveness of our best-performing LLM fine-tuned using LoRA with RAG. Our RAG setup consists of retrieving up to 16 examples from the training set that are closely related to the input prompt, which is similar to other approaches previously proposed for various SE tasks [39, 41, 70].
- RQ4: **Can we enhance the effectiveness of LLMs for code generation in the APPS dataset using LoRA and QLoRA?** Lastly, we explore whether LLM fine-tuned using LoRA and QLoRA show improvement in functional correctness in the APPS dataset. We fine-tune our best-performing LLM using LoRA and QLoRA on the training set of APPS, and report the average of test cases passed as well as the Pass@k on APPS' test set for introductory, interview, and competition-level coding problems.

4.2 Datasets 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 [9, 10, 48, 50, 61] 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 [9] have extensively been used to benchmark code generation approaches [3, 9, 82]. While HumanEval is widely utilized, it lacks a training corpus to evaluate fine-tuning or PEFT approaches. As our study's focus is on specializing LLMs using PEFT techniques, we have opted not to utilize HumanEval. Instead, we choose to use three other widely-used code generation datasets: the Conala [87], CodeAlpaca [8], and APPS [22] datasets. All

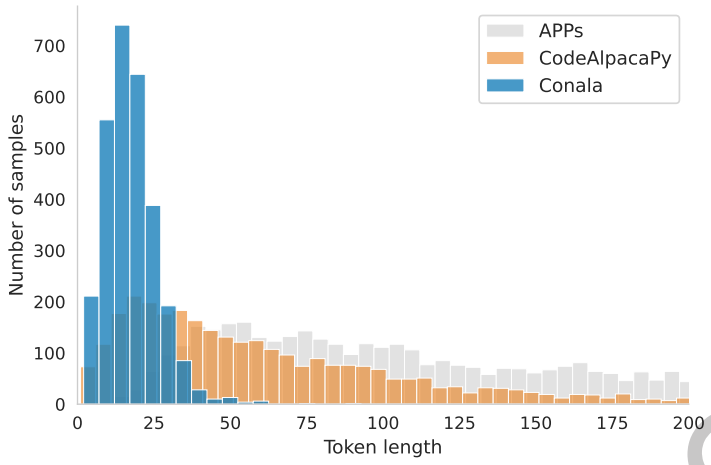


Fig. 2. Token length distribution of the Conala, CodeAlpacaPy, and APPS datasets.

datasets provide an ample number of examples that can be employed for fine-tuning a model and have been used in prior code generation studies with LLMs [71, 73, 88, 91].

Conala dataset. We use a curated version of the Conala dataset [91]. 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 the first example in Table 2, providing more context to the model for generating relevant code. In Figure 2, we report the token length distributions of the three datasets. In Conala, most code solutions are short and one-liners, making it relatively easy for an LLM to generate exact match predictions. In this curated version of the dataset, the authors ensured that each sample in the validation and test sets contained 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. The dataset contains 2,135/201/543 samples as the training/validation/test sets, respectively.

CodeAlpacaPy dataset. We construct a curated Python version of the CodeAlpaca [8] dataset by specifically selecting the Python data samples within the CodeAlpaca dataset. We filter out code samples that cannot be statically parsed to ensure the dataset encompasses only syntactically valid Python codes. As illustrated in the bottom example of Table 2 and in Figure 2, CodeAlpacaPy contains lengthier and more complex examples than Conala, allowing for a more comprehensive evaluation of PEFT for code generation. The dataset contains 2,192/314/628 samples as the training/validation/test sets, respectively.

APPS dataset. The APPS dataset consists of 10,000 code generation problems, each paired with Python solutions. These problems are categorized into three difficulty levels: introductory, interview, and competition, with solutions varying from simple one-liners to complex algorithms. We can see in Fig. 2 and Table 2 that APPS include more lengthy and complex examples than the two other datasets. On average, each problem is accompanied by 21.2 test cases, designed to evaluate the functional correctness of the generated code. The original dataset is split into 5,000 samples for training and 5,000 for testing. In this study, we use 4,500 samples for training, 500 for validation, and 750 for testing, ensuring a balanced distribution of 250 test samples per difficulty level.

Table 2. Overview of the code generation task, with three examples taken from the Conala, CodeAlpacaPy, and APPS datasets.

	Conala
Prompt:	### Instruction: *map two lists 'keys' and 'values' into a dictionary* ### Response:
Ground truth:	`dict([(k, v) for k, v in zip(keys, values)])`
	CodeAlpacaPy
Prompt:	### Instruction: *Write a function to calculate the standard deviation of data points in Python.* ### Response:

Ground truth:	`def stdev(data):`
	`  avg = sum(data) / len(data)`
	`  total = 0`
	`  for x in data:`
	`  total += (x - avg) ** 2`
	`  return (total / (len(data) - 1)) ** 0.5`

	APPS
Prompt:	### Instruction: *You are given a string s = s1 s2 . . . sn of length n, which only contains digits 1, 2,..., 9. A substring s[l...r] of s is a string slsl+1sl+2 ...sr. A substring s[l...r] of s is called even if the number represented by it is even. Find the number of even substrings of s. Note, that even if some substrings are equal as strings, but have different l and r, they are counted as different substrings. The first line contains an integer n (1 ≤ n ≤ 65000) — the length of the string s. The second line contains a string s of length n. The string s consists only of digits 1, 2,..., 9. Print the number of even substrings of s.* ### Response:

Ground truth:	`n = int(input())`
	`ans = 0`
	`for i in range(n):`
	`  for j in range(i, n):`
	`  if int(s[i:j+1]) \% 2 == 0:`
	`  ans += 1`
	`print(ans)`

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 "### Response:" play the role of delimiting the instruction, *i.e.*, natural language intent, and the answer, *i.e.*, code generation. Note that this prompt design may not be optimal, but this kind of instruction template has shown to be effective in prior works [36, 89]. 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 by generating up to 10 code candidates.

4.3 ICL and RAG

We conduct experiments using ICL and RAG on the Conala and CodeAlpacaPy datasets. For both techniques, we select the maximum number of samples that can fit into our GPU memory. For ICL, we use up to 16 examples for the Conala dataset and 8 examples for CodeAlpacaPy. These examples are randomly sampled from the corresponding training datasets and concatenated with the input prompt during inference. For RAG, we leverage GTE-small, a general-purpose, lightweight embedding model that outperforms many larger models, including OpenAI's proprietary embeddings [34]. We generate embeddings for all instructions (excluding the code) in the training sets. At inference time, we retrieve up to 16 examples for Conala and 4 examples for CodeAlpacaPy, selecting those with instructions most similar to the test input. As with ICL, the retrieved examples are concatenated with the input problem to guide code generation.

4.4 Small and Large Language Models

In order to carry out a comprehensive analysis, we selected our SLMs 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 have been pre-trained using open-source data. Secondly, we selected LLMs, which have been released within the past two years. 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 SLMs, and the others as LLMs. Note that we selected models that fit a single 24GB GPU for fine-tuning and inference without causing memory overflow. In total, we included 11 SLMs and LLMs from diverse families of models to conduct our experiments.

- **SLMs**. We use CodeGen-350M-mono [48], CodeT5+-220M [71], and CodeT5+-770M [71] as SLMs. CodeGen-350M-mono is an autoregressive language model and a small version of CodeGen pre-trained on various programming languages and further fine-tuned on Python data. CodeT5+-220M and CodeT5+-770M are encoder-decoder language models that improve upon CodeT5 by leveraging a two-staged pre-training phase on natural language and code data, and new learning objectives.
- **CodeGen2** [47] is a family of prefix-based language models which combines the learning schemes of a bi-directional encoder and a uni-directional decoder. CodeGen2 improves upon CodeGen [48], therefore we do not include the CodeGen family in our evaluation. CodeGen2 models were pre-trained on a deduplicated version of TheStack [28] spanning a wide range of languages. We employ CodeGen2-1B, CodeGen2-3.7B and CodeGen2-7B.
- **CodeLlama** [56] is a family of LLMs based on Llama 2 [64]. Each model was initialized with Llama 2 and further pre-trained on code. CodeLlama comes in three different variants: CodeLlama specialized for code, CodeLlama-Instruct specialized for instruction-tuning and CodeLlama-Python specialized for Python. We

employ CodeLlama-7B, CodeLlama-7B-Instruct and CodeLlama-7B-Python to initiate our experiments. In RQ4, we fine-tune CodeLlama-13B-Python and CodeLlama-34B-Python using QLoRA.

4.5 Metrics

We measure the effectiveness of the models through widely used metrics in prior code generation work. For experiments on Conala and CodeAlpacaPy, we report the Exact Match (EM) and CodeBLEU [55] metrics. Given a generated code and a ground truth, the EM returns 1 if both codes are identical, otherwise 0. 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. For our experiments on the APPS dataset, we report two metrics: the average number of test cases passed and Pass@k. The average number of test cases passed evaluates how well the model performs by measuring the proportion of test cases that its generated code passes for each sample. In contrast, Pass@k is a more stringent metric that measures the percentage of problems for which at least one of the top k generated code samples passes all test cases, reflecting the model's ability to produce fully correct solutions within k attempts.

4.6 Implementation Details

For all our experiments, we used a single NVIDIA RTX A5000 24GB GPU. We study a total of seven tuning techniques: Full fine-tuning, ICL, LoRA [24], IA3 [37], Prompt tuning [30], Prefix tuning [33], and QLoRA [13]. We implemented all the tuning techniques using HuggingFace [79] and PEFT [43] libraries.

We used full fine-tuning only for the SLMs, as tuning all the parameters of the LLMs is computationally intractable within a maximum GPU memory of 24GB. We set the learning rate to $5e-5$. For LoRA and IA3, we applied the low-rank matrix decomposition on the attention layers of the models and set $r = 16$ and $\alpha = 32$. For implementing QLoRA, we use 8-bit and 4-bit quantization [12]. We set the learning rate to $3e-4$ for LoRA, IA3 and QLoRA. For Prompt tuning and Prefix tuning, we prepended a set of 20 trainable continuous virtual tokens to each input sample of the models and applied learning rates of $3e-3$ and $3e-2$.

We used Adafactor [60] optimizer with 16-bit float precision for all models. We fine-tuned the models for a maximum of five epochs and evaluated them every $0.2 * len(train\_set)$ optimization steps. We fine-tune all models with a batch size of 8. We selected the checkpoint with the lowest evaluation loss for inference and found that beam search with a beam size of 10 yields the best effectiveness. Given the various token length distribution and complexity of the datasets, we generate codes with up to 64, 128, and 1024 tokens for Conala, CodeAlpacaPy, and APPS, respectively. We make our code publicly available: https://github.com/martin-wey/peft-llm-code.

5 EXPERIMENTAL RESULTS

5.1 RQ1: Baseline Effectiveness of Models Using Zero-Shot and ICL

We start by investigating the baseline effectiveness of all SLMs and LLMs for match-based code generation. Specifically, we use zero-shot and ICL approaches with up to 16 retrieved random examples for the Conala dataset and eight for the CodeAlpacaPy dataset. The reason behind utilizing fewer examples for CodeAlpacaPy is because considering 16 examples results in out-of-memory errors under our setup. We evaluate the models' effectiveness using EM@10 and compare them across these two datasets in Fig. 3. Note that CodeGen2 architecture results in substantially more GPU memory usage than other models, which explains why we evaluate ICL with fewer examples than other models.

First, we observe a substantial gap in EM@10 between the two datasets. This difference can be explained by the fact that the CodeAlpacaPy dataset contains much more challenging samples compared to the Conala dataset, as shown in Table 2.

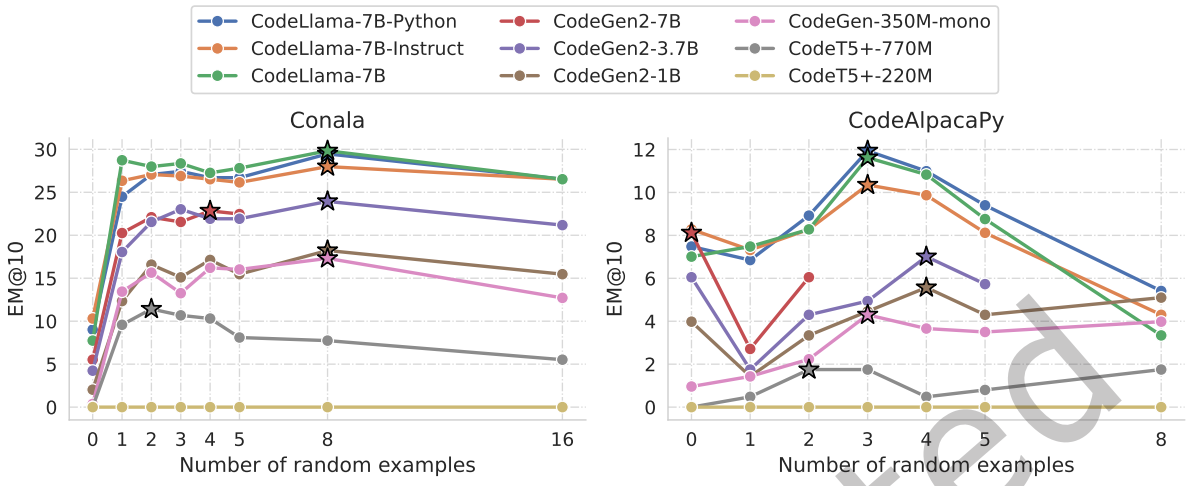


Fig. 3. [RQ1] – Effectiveness of the models using ICL with various number of random examples on the Conala and CodeAlpacaPy datasets.

Second, there is a notable gap in effectiveness between SLMs and LLMs, regardless of the number of examples provided. This observation highlights the advantages of large-scale pre-training and the use of larger models in this context.

For the Conala dataset, increasing the number of examples leads to higher EM@10 scores. However, when using more than eight examples, the effectiveness of the models begins to decline. For the CodeAlpacaPy dataset, a similar trend is observed, but the optimal number of examples is smaller. Most models achieve their best EM@10 scores when using three or four examples. This observation underscores the limitation of ICL, as adding more examples results in a degradation of the models' effectiveness.

Finally, CodeLlama models outperform all models across both datasets, achieving a peak EM@10 of 29.83 on Conala (CodeLlama-7B) and 11.94 on CodeAlpacaPy (CodeLlama-7B-Python). In contrast, smaller models, such as CodeGen2-3.7B achieves an EM@10 of 23.94 and 7.00 on Conala and CodeAlpacaPy, respectively.

Answer to RQ1: ICL drastically improves the effectiveness of all models compared to zero-shot. Our best model, CodeLlama-7B, achieves EM@10 scores of 29.83 (7.73) and 11.62 (7.01) on Conala and CodeAlpacaPy with ICL (zero-shot), respectively.

5.2 RQ2: Effectiveness of Models using PEFT Techniques

We report the detailed results of the effectiveness of the SLMs and LLMs on match-based code generation for both Conala and CodeAlpacaPy datasets in Table 3.

SLMs vs. LLMs. CodeGen-350M-mono with LoRA demonstrates the best effectiveness on average among small models, while CodeLlama-7B-Python with LoRA is the best LLM on average. Under the same 24GB GPU memory limitation, the best LLM surpasses the best small model by 39.8%, 41.7%, and 47.1% (72.3%, 48.8%, and 9.1%) in EM@1, EM@10, and CodeBLEU concerning the Conala (CodeAlpacaPy) dataset, respectively.

SLMs. Among the SLMs, CodeGen-350M-mono shows the highest effectiveness across all metrics on both datasets. Our results align with prior studies [48, 73, 91] that identified CodeGen-350M-mono as a robust SLM for

Table 3. [RQ2] – Comparison of the SLMs and LLMs using various tuning techniques (blue : best-performing tuning method per model, orange : best overall performing model).

Model	Tuning	# Params	Conala			CodeAlpacaPy		
			EM@1	EM@10	CodeBLEU	EM@1	EM@10	CodeBLEU
SLMs								
CodeT5+-220M	Full FT	220M	3.87	8.84	16.70	**3.98**	**7.64**	**25.49**
	LoRA	2.7M	**6.08**	**12.71**	**18.96**	3.34	5.26	24.32
	IA3	0.17M	4.42	10.68	17.08	0.64	1.27	22.06
	Prompt tuning	0.03M	4.79	9.21	16.30	0.96	2.07	19.01
	Prefix tuning	0.18M	3.13	7.55	14.56	0.16	1.27	20.80
CodeT5+-770M	Full FT	770M	4.05	8.29	15.11	3.19	6.21	27.73
	LoRA	7M	**8.66**	17.13	**20.64**	**3.66**	**6.85**	**26.10**
	IA3	0.4M	8.10	**17.50**	18.68	2.87	5.26	25.84
	Prompt tuning	0.04M	7.37	15.47	16.75	1.91	3.82	20.57
	Prefix tuning	0.5M	4.97	11.97	16.77	0.16	1.27	22.91
CodeGen-350M-mono	Full FT	350M	7.92	18.42	14.68	2.23	5.73	21.78
	LoRA	1.3M	**12.52**	25.60	17.89	**4.62**	**10.70**	**30.09**
	IA3	0.16M	11.42	**25.78**	**18.83**	4.46	**10.70**	28.56
	Prompt tuning	0.02M	7.92	20.26	16.29	0.0	0.0	25.91
	Prefix tuning	0.4M	5.34	12.52	17.53	0.0	0.0	26.89
LLMs								
CodeGen2-1B	LoRA	2M	9.39	**23.02**	**19.76**	**3.82**	9.08	23.48
	IA3	0.2M	10.13	22.84	18.64	**3.82**	**9.87**	**24.42**
	Prompt tuning	0.04M	**11.97**	22.65	18.38	0.80	2.07	18.17
	Prefix tuning	0.6M	5.89	15.84	18.46	0.0	0.32	13.68
CodeGen2-3.7B	LoRA	4M	**11.60**	25.97	19.00	**5.41**	10.70	23.75
	IA3	0.5M	10.87	25.23	19.21	**5.41**	**10.99**	**26.26**
	Prompt tuning	0.08M	11.05	**26.89**	19.53	0.0	0.0	23.42
	Prefix tuning	1.3M	10.68	24.68	**20.23**	0.16	0.32	21.73
CodeGen2-7B	LoRA	8.3M	11.23	**29.83**	**23.86**	5.57	11.94	27.73
	IA3	1M	11.42	29.65	21.98	**5.73**	**12.42**	**28.26**
	Prompt tuning	0.08M	**11.97**	27.26	22.37	0.0	0.0	25.40
	Prefix tuning	2.6M	9.95	23.94	22.29	0.0	0.32	25.72
CodeLlama-7B	LoRA	12.5M	**20.07**	**39.31**	**25.33**	7.33	**16.24**	**32.05**
	IA3	1M	17.68	37.20	23.19	**8.12**	15.45	30.47
	Prompt tuning	0.08M	19.15	38.12	25.01	0.32	0.48	31.55
	Prefix tuning	2.6M	8.47	19.52	23.19	0.16	0.16	28.09
CodeLlama-7B-Instruct	LoRA	12.5M	17.68	**36.28**	24.27	7.01	**17.04**	**31.42**
	IA3	1M	15.84	36.10	24.71	**8.12**	16.72	31.01
	Prompt tuning	0.08M	**18.97**	35.54	**25.77**	1.59	3.50	31.14
	Prefix tuning	2.6M	10.13	18.23	23.66	0.64	0.96	31.27
CodeLlama-7B-Python	LoRA	12.5M	**17.50**	36.28	24.27	7.96	15.92	32.84
	IA3	1M	14.55	31.12	24.74	**8.76**	**16.56**	29.82
	Prompt tuning	0.08M	16.76	**37.02**	**26.31**	0.96	3.03	**33.46**
	Prefix tuning	2.6M	9.76	22.47	19.47	0.0	0.0	30.71

Python code generation tasks. Interestingly, although it requires tuning approximately 1% of the total parameters

of the model, LoRA appears as the best tuning technique, surpassing full fine-tuning by a considerable margin across nearly all configurations. For instance, the EM@10 score for CodeGen-350M-mono on the Conala dataset, with full fine-tuning, is 18.42, while it soars to 25.60 with LoRA.

LLMs. In Figure 4, we present a comparative analysis of the models' effectiveness when tuned using LoRA, focusing on CodeBLEU and EM@10 scores. Both plots clearly establish CodeLlama models as the best-performing LLMs in our study. Remarkably, CodeGen2-7B, despite sharing a similar number of parameters, lags behind all CodeLlama-7B variants. Unsurprisingly, harnessing larger models leads to better effectiveness. Given the low computational costs of PEFT techniques, leveraging smaller models in a context akin to ours seems counter-productive. Subsequently, in this paper, we demonstrate that even larger models can be fine-tuned through the combination of PEFT with quantization.

Best PEFT technique. Overall, LoRA emerges as the most effective PEFT technique among the studied ones. Although being presented as an incremental improvement over LoRA [37], IA3 often shows lower scores compared to LoRA. Prompt tuning appears as another viable tuning option, while further reducing the number of trainable parameters. However, Prefix tuning fails to effectively adapt the larger models to both datasets.

Our analysis reveals notably higher EM scores for the Conala dataset, which can be attributed to differences in task complexity between the two datasets (see Section 4.2). It is important to note that CodeBLEU scores on Conala are comparatively lower due to the metric's reliance on dataflow graph computations, which may not always be available for small code examples.

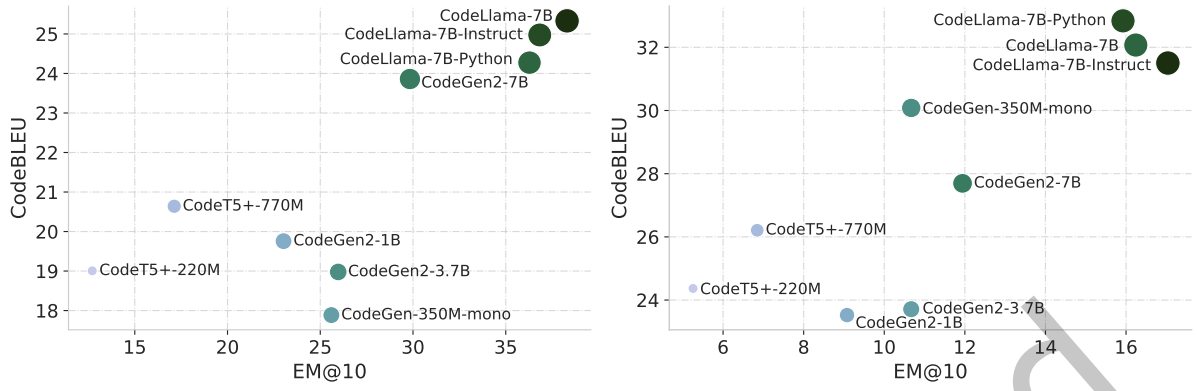
Effect of quantization with QLoRA. We explore the potential benefits of employing QLoRA [13], a computationally efficient technique that combines LoRA with 8-bit or 4-bit quantization for fine-tuning LLMs. In Figure 5, we display EM@10 scores for three CodeLlama model variants: CodeLlama-7B-Python, CodeLlama-13B-Python, and CodeLlama-34B-Python, alongside peak GPU memory consumption consistently below 24GB for each tuning configuration. The results underscore a significant improvement in the effectiveness of larger quantized models on Conala, with a more moderate impact on CodeAlpacaPy. For instance, CodeLlama-34B-Python, fine-tuned with QLoRA-4bit, achieves a substantial 12.2% increase in Conala's EM@10 score (40.70) compared to CodeLlama-7B-Python with LoRA (36.28). Surprisingly, QLoRA also brings notable improvements over LoRA for CodeLlama-7B-Python on Conala, while achieving comparable results on CodeAlpacaPy. The application of quantization enables the utilization of larger models that can be accommodated within a single 24GB GPU. Specifically, for CodeLlama-7B-Python, QLoRA-4bit achieves a remarkable 2x reduction in peak memory usage while significantly improving the EM@10 score.

Answer to RQ2: LLMs with PEFT consistently and significantly outperform SLMs under the same GPU limit. Specifically, the best-performing LLM with PEFT surpasses the best small model by 39.8–72.3% in terms of EM@k. Among different PEFT techniques, LoRA is the most effective. In addition, applying quantization with LoRA results in a drastic decrease in GPU usage while maintaining effectiveness on both datasets and accommodating the fine-tuning of larger models up to 34B parameters.

5.3 RQ3: Comparative Analysis of LoRA, ICL, and RAG

In this RQ, we aim to investigate whether PEFT techniques consistently outperform the widely used ICL and RAG when applying LLMs in match-based code generation.

In Figure 6, we compare the effectiveness of the SLMs and LLMs using ICL and LoRA in terms of CodeBLEU and EM@10. In this figure, we report the highest metrics achieved over the different ICL configurations for each model. In Figure 7, we explore the effectiveness of CodeLlama models using RAG, with up to 16 and 4 retrieved



(a) Conala (b) CodeAlpacaPy

Fig. 4. [RQ2] – Effectiveness of the models fine-tuned using LoRA for both datasets in terms of EM@10 and CodeBLEU.

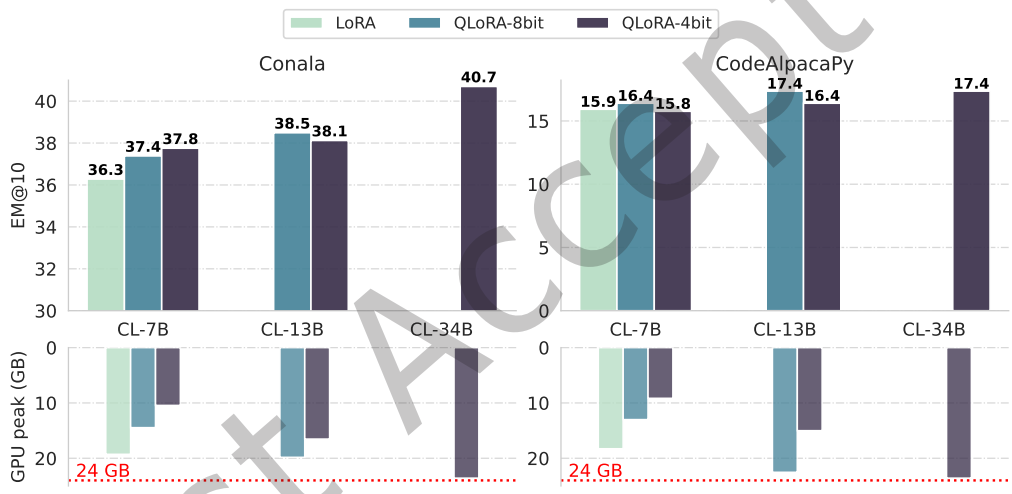


Fig. 5. [RQ2] – Effectiveness and GPU usage of 7B, 13B, and 34B CodeLlama-Python (CL) LLMs fine-tuned using LoRA and QLoRA with 8-bit and 4-bit quantization.

examples for Conala and CodeAlpacaPy, respectively. Similar to RQ1, we use fewer examples for CodeAlpacaPy to avoid out-of-memory errors. We compare the effectiveness of RAG with LoRA and the best EM@10 score achieved using ICL.

LoRA vs. ICL. As shown in Fig. 6, all models fine-tuned with LoRA demonstrate significantly higher EM@10 scores compared to ICL across both datasets. For example, CodeLlama-7B-Python with LoRA tuning achieves a 23.1% improvement in EM@10 on Conala (36.28 for LoRA vs. 29.47 for ICL). This pattern holds for CodeAlpacaPy, with even greater relative gains in EM@10. However, we observe some variation in CodeBLEU scores for most models on CodeAlpacaPy. For instance, CodeLlama-7B sees a CodeBLEU increase of 2.36 with LoRA. On CoNala, though, the impact of LoRA on CodeBLEU is less pronounced than that of ICL. These differences can be explained by the nature of the metrics: EM@10 is more conservative, requiring the generated solution to exactly match the

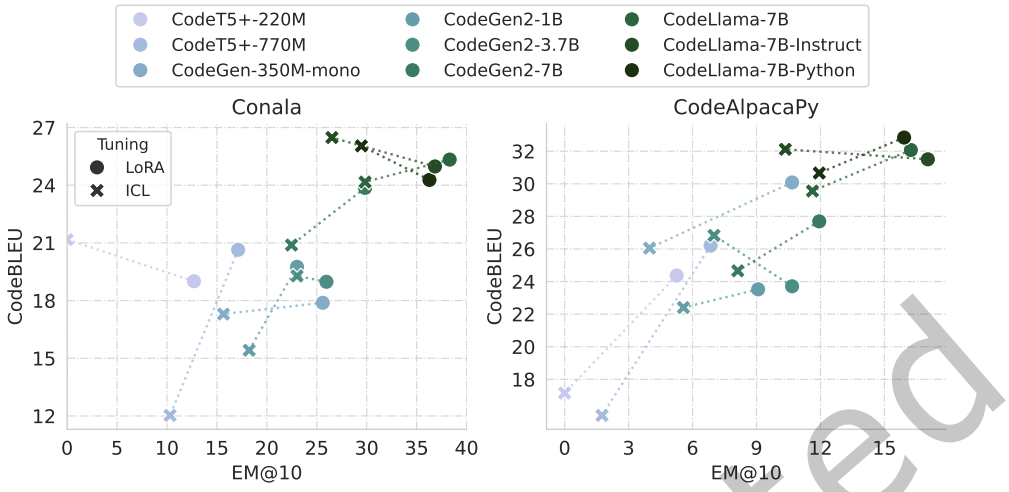


Fig. 6. [RQ3] – Comparison of the effectiveness of the models fine-tuned using LoRA and ICL on the Conala and CodeAlpacaPy datasets.

ground truth, while CodeBLEU gives higher scores for solutions that are close but not exact. This distinction highlights how LoRA better adapts models to downstream datasets, particularly when precision is crucial.

RAG vs. ICL vs. LoRA. In comparing RAG, ICL, and LoRA on the CoNala dataset, RAG demonstrates higher effectiveness than ICL but falls short of LoRA's effectiveness across all three CodeLlama model variants. Notably, CodeLlama-7B achieves a maximum of 29.83 and 35.17 EM@10 with ICL and RAG, respectively, whereas the model tuned with LoRA reaches an EM@10 of 39.31.

For both Conala and CodeAlpacaPy datasets, the gains in EM@10 get thinner as we increase the number of examples using RAG. EM@10 saturates at around 8–16 examples for Conala and 3–4 examples for CodeAlpacaPy. Furthermore, we note that for the more challenging CodeAlpacaPy datasets, RAG yields lower EM@10 compared to randomly selected examples using ICL, highlighting RAG's limitations when problem complexity increases. LoRA, however, consistently outperforms both RAG and ICL on CodeAlpacaPy, highlighting its superior ability to adapt to more challenging datasets.

Answer to RQ3: LoRA is superior to ICL and RAG on Conala and CodeAlpacaPy datasets across the three CodeLlama-7B variants.

5.4 RQ4: Exploration of LoRA and QLoRA for Code Generation on APPs

In this final RQ, we explore the broader applicability of LoRA and QLoRA, to enhance CodeLlama-7B-Instruct's effectiveness for execution-based code generation. The reason for choosing the instruct variant of CodeLlama-7B is because the model generally shows higher effectiveness than the other model variants on APPs in the seminal paper of CodeLlama [56]. We do not compare LoRA and QLoRA with ICL and RAG for this dataset because they require increasing the prompt length beyond 2,048 tokens, which leads to out-of-memory errors. Our results, summarized in Table 4, focus on the average number of test cases passed (Avg) and Pass@k for introductory, interview, and competition-level tasks.

For both introductory and interview-level code generation tasks, LoRA and QLoRA-8/4bit lead to significant improvements in the average number of passed test cases. Specifically, QLoRA-4bit results in a notable 52%

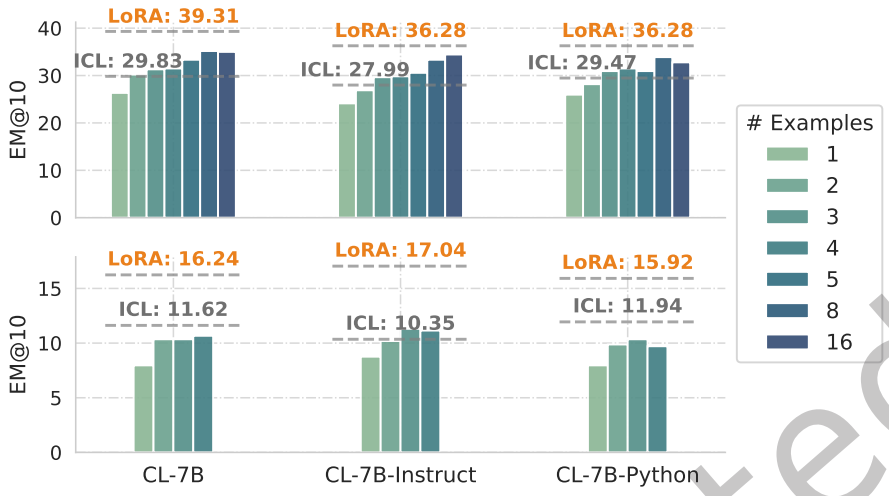


Fig. 7. [RQ3] – Comparison of the effectiveness of RAG with various number of retrieved examples against ICL and LoRA on the Conala (top) and CodeAlpacaPy (bottom) datasets. The ICL scores depict the highest scores achieved for each model in RQ1.

Table 4. [RQ4] – Effectiveness of CodeLlama-7B-Instruct on the APPs dataset in zero-shot and using LoRA, QLoRA-8bit, and QLoRA-4bit in terms of average passed tests (Avg) and Pass@k (P@k).

	Introductory				Interview				Competition			
Model	Avg	P@1	P@2	P@5	Avg	P@1	P@2	P@5	Avg	P@1	P@2	P@5
CodeLlama-7B-Instruct	13.66	4.16	6.24	8.80	13.44	0.80	1.32	2.40	6.27	**0.56**	**1.00**	**2.00**
+LoRA	19.57	5.60	8.04	11.20	16.96	**1.04**	**1.80**	**3.20**	6.93	0.32	0.60	1.20
+QLoRA-8bit	17.63	3.68	5.40	7.60	15.53	**1.04**	1.64	2.40	**7.59**	0.24	0.48	1.20
+QLoRA-4bit	**20.84**	**5.76**	**8.40**	**12.40**	**20.34**	**1.04**	1.76	2.80	6.66	0.48	0.88	1.60

increase in the average number of tests passed compared to the base model. In terms of Pass@k metrics, both LoRA and QLoRA-4bit demonstrate gains at the introductory level, with Pass@5 improving by +3.60% over the base model. However, these improvements are less substantial for interview and competition-level code generation, reflecting the greater complexity and challenge posed by these more advanced tasks.

Answer to RQ4: LoRA and QLoRA enhance CodeLlama-7B-Instruct's effectiveness on APPs, particularly at the introductory, with QLoRA-4bit boosting the average number of passed test cases by 52% and Pass@5 by 40%. However, improvements are less notable for interview and competition-level tasks.

6 DISCUSSION

Our study explores PEFTs applied to code LLMs, elucidating the positive impact of these applications in efficiently tuning LLMs to task-specific datasets for code generation. In particular, our study illustrates the practicality of

fine-tuning LLMs using PEFT, thereby alleviating the dependence of practitioners on large and expensive infrastructures. Our findings also pinpoint several promising areas for future exploration, including the investigation of efficient techniques across diverse fine-tuning settings, during inference, and for other SE tasks.

Efficient techniques for LLMs of code. Our work emphasizes efficient fine-tuning techniques, democratizing the tuning of LLMs to a broad audience. Nonetheless, our study did not include the exploration of efficient techniques for low-cost inference. While PEFT techniques require additional fine-tuning time compared to ICL and RAG, it is noteworthy that these techniques do not impose any supplementary time cost during inference. Nonetheless, we acknowledge the necessity of future investigations into techniques to reduce the time cost associated with LLMs during inference.

PEFT and ICL/RAG are non-exclusive techniques that can be used jointly. However, we decided not to include experiments on the application of ICL/RAG to LLMs fine-tuned using PEFT. In practice, increasing the number of ICL/RAG examples at inference entails increased computational overhead as the token length of the prompt expands. Consequently, we contend that employing ICL/RAG on a fine-tuned LLM might counterproductively escalate computational demands, outweighing potential benefits.

From a different angle, prior studies [18, 78, 83] highlighted the need to consider pre-trained language models and LLMs of code in continual learning settings. In this paradigm, the model must dynamically adapt to new data over time while preserving performance on previously seen data. In the specific setting of continuously evolving LLMs, PEFT techniques potentially offer valuable benefits. Nonetheless, it is yet to be determined whether PEFT techniques can efficiently adapt LLMs under a continual learning setting for code-related tasks, without compromising the retention of past knowledge.

Effectiveness of QLoRA. Across all study datasets, we observed that QLoRA-4bit demonstrated competitive or comparable effectiveness to other PEFT methods. Notably, QLoRA-4bit outperformed LoRA and QLoRA-8bit on the Conala and APPs datasets. We hypothesize that this improvement stems from the regularization effect of reducing weight precision to 4 bits, which helps stabilize fine-tuning and mitigates overfitting. These findings highlight the potential for more efficient PEFT techniques, though further exploration is needed to fully understand their broader applicability.

New findings for PEFT in software engineering. Our findings in RQ1 reveal that PEFT methods outperform full fine-tuning for SLMs in code generation tasks. This stands in contrast to prior large-scale studies in NLP, such as Ding et al. [14], which demonstrated the superior effectiveness of full fine-tuning over techniques like LoRA, Prompt Tuning, and Prefix Tuning across a wide range of NLP tasks.

In the context of software engineering, while previous studies [38, 40] have shown that PEFT methods, like LoRA, can perform comparably to full fine-tuning for SLMs, our results go further. We show that all PEFT techniques studied in this paper significantly outperform full fine-tuning for SLMs like CodeGen-350M-mono and CodeT5+-770M on the Conala and CodeAlpacaPy datasets (see Table 3), highlighting the clear advantages of PEFT in these scenarios. However, due to resource constraints, we were unable to evaluate full fine-tuning for LLMs, leaving room for future studies to explore this further in the software engineering domain.

Additionally, our research uncovers new insights into the benefits of QLoRA and the comparative effectiveness of LoRA versus RAG for code generation tasks. First, in RQ3 and RQ4, we demonstrate that QLoRA offers comparable or even superior performance to LoRA while drastically cutting computational costs. Second, we reveal limitations of ICL and RAG, showing that LLM effectiveness tends to plateau as more examples are retrieved. In contrast, our study highlights the consistent advantages of PEFT techniques like LoRA and QLoRA in overcoming these limitations.

SE tasks and multi-tasking. To ensure a focused study, we avoided adding extra tasks and datasets, preventing an excessively broad set of analyses. Exploring PEFT techniques for LLMs across varied tasks and datasets is a

promising direction for future research. In particular, Lorahub [26], a recently introduced framework for multi-task learning, demonstrates that a composition of LoRA modules trained on different tasks can generalize to new, unseen tasks while offering a strong performance-efficiency trade-off. We believe applying similar approaches in AI for SE holds great potential, particularly as the research field aims at automating a broad range of code-related tasks.

7 THREATS TO VALIDITY

External validity. One main threat relates to the choice of our SLMs and LLMs. We mitigated this threat by carefully selecting a diverse set of models, as explained in Section 4.4. These models encompass various families of LLMs, trained on distinct pre-training data and learning objectives, and varying in size. Furthermore, we did not select larger model variants except when using QLoRA, as other PEFT techniques, ICL, and RAG limit the use of larger models within our resource constraints.

Another external threat to the validity is related to the quality and representativeness of the fine-tuning datasets. 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 [49, 73, 91] on code generation tasks. Furthermore, the authors enriched each natural language intent with hints, enhancing the alignment of input prompts with possible human intents. To enrich our study, we included CodeAlpacaPy as a second dataset which encompasses lengthier examples, bringing another line of analysis. We did not include evaluation datasets such as HumanEval [9] and MBPP [3], as they do not include training examples. However, to further expand our study, we explored the effectiveness of LoRA and QLoRA for execution-based code generation on the APPs dataset.

Finally, the monolingual aspect of our datasets constitutes another threat to external validity. We studied full fine-tuning, PEFT, ICL, and RAG for code generation of Python code snippets. However, we anticipate that PEFT is also applicable to other programming languages, considering the impressive generation capabilities of LLMs on a diverse range of programming languages [2, 6].

Internal validity. The hyperparameter choices for the PEFT methods constitute the main threat to internal validity. For each PEFT technique, we used hyperparameters values which have been used in previous work on PEFT for code models as well as in the seminal papers that contributed the PEFT techniques. Additionally, since LoRA with $r = 16$ and $\alpha = 32$ consistently outperforms all configurations of ICL and RAG across our top three models, conducting a detailed hyperparameter sensitivity analysis of LoRA could further solidify the advantage of PEFT over ICL and RAG. Future work could explore the sensitivity of key LoRA hyperparameters, such as rank r and scaling factor α, across a broader range of software engineering tasks.

Construct validity. The choice of our evaluation metrics constitutes the main threat to construct validity. To mitigate this threat, we selected evaluation metrics widely used in prior works [22, 32, 42, 56, 72, 84] on code generation. Furthermore, we evaluate each approach using EM@k on Conala and CodeAlpacaPy, which enriched our analysis by computing the exact match over different ranges of code candidates. Similarly, for APPs, we evaluate the base model and LoRA/QLoRA on Pass@k with up to 5 candidates. Finally, we did not use Pass@k metrics as the CoNaLa and CodeAlpacaPy datasets do not include unit tests. Enriching the datasets with unit tests constitutes an interesting area of future work.

8 RELATED WORK

In this section, we overview existing work on LLMs for code generation and contrast previous contributions on efficient model adaptation of code for downstream tasks with our study.

Automated Code Generation. A significant portion of code generation techniques [1, 4, 21, 63, 72] relies on deep-learning-based approaches. The latest trend in automated code generation revolves around leveraging LLMs like GPT models [50] due to their remarkable breakthroughs in this domain. One notable example is Codex, developed by Chen et al. [9], which is a fine-tuned version of GPT-3. Other noteworthy models following the success of Codex include CodeGen [48], CodeGen2 [47] and CodeLlama [56]. These LLMs effectively democratize the breakthrough performance achieved by Codex and bring it to a broader audience. However, the high computational costs associated with full fine-tuning for LLMs to achieve optimal performance are impractical for most researchers and practitioners. We believe that our study can shed light on more efficient and cost-effective approaches to fine-tuning these LLMs, mitigating the computational burdens associated with their adoption.

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 rendering the fine-tuning computation costs low, *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 [53, 80] 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 an LLM. Prior works [46, 91] designed selection techniques to retrieve highly relevant prompt examples tailored to downstream tasks, outperforming random selection methods. Lastly, recent research [61] 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. Instead, we opted for a simple implementation of ICL by selecting random few-shot examples using different seeds. Expanding this study to incorporate more ICL approaches would enhance the comparison with PEFT techniques for code.

Regarding PEFT techniques, prior research in code intelligence has focused on Prompt tuning [30], Prefix-tuning [33] and Adapters [20, 23, 25, 57, 58]. Wang et al. [68] 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. Goel et al. [20] explored the use of programming-language-specific adapters for knowledge transfer in pre-trained language models, demonstrating that tuning BERT with these adapters surpass CodeBERT on cloze test and code clone detection. Choi et al. [10] designed a code-specific Prefix tuning approach within a sequence-to-sequence architecture for generation tasks. Our study differs from these three previous works as they focus on SLMs, whereas we propose the first comprehensive study of PEFT techniques with LLMs for code generation. Moreover, our study includes LoRA, IA3, and QLoRA, which none of the previous work in code intelligence considered for efficiently tuning LLMs of code. Wang et al. [69] showcased the superiority of utilizing Adapters for fine-tuning pre-trained language models over full fine-tuning. Recent work have contributed empirical studies for various software engineering tasks, including code change [40], code summarization [38, 57], defect prediction [38], and code clone detection [57], using Adapter tuning and LoRA for SLMs. Our research diverges from these prior work, as we concentrate on LLMs. Although we did not incorporate Adapters in our investigation, we believe that LoRA, IA3, Prompt tuning, Prefix tuning, and QLoRA provide a sufficiently thorough analysis of PEFT techniques. We recognize the value of exploring additional PEFT techniques for various code intelligence tasks in the future.

9 CONCLUSION AND FUTURE WORK

This study establishes the effectiveness of PEFT techniques in fine-tuning LLMs for code generation. Our comparative analysis across various parameter-efficient techniques, including LoRA, IA3, Prompt tuning, Prefix tuning, and QLoRA, reveals the superiority of PEFT over full fine-tuning for SLMs and ICL and RAG for LLMs. Furthermore, our study illustrates the practicality of PEFT under a limited resources scenario, effectively mitigating the reliance on large and expensive computational infrastructures. To the best of our knowledge, this study is among the first comprehensive exploration of PEFT techniques for LLMs in software engineering, suggesting a promising avenue for future research. We anticipate our findings will inspire further investigation into the application of PEFT techniques in software engineering, with potentially far-reaching impacts. Our future work will extend the study to alternative software engineering tasks such as automated code review and comment generation. Finally, we aim to validate further relevance of PEFT techniques under multi-tasking and continual learning settings for automated software engineering.

REFERENCES

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

[2] 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).

[3] 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).

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

[5] 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.

[6] 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).

[7] 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.

[8] Sahil Chaudhary. 2023. Code Alpaca: An Instruction-following LLaMA model for code generation. https://github.com/sahil280114/codealpaca.

[9] 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).

[10] 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.

[11] 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).

[12] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. *arXiv preprint arXiv:2208.07339* (2022).

[13] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314* (2023).

[14] 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).

[15] 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.

[16] 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).

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

[21] 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).

[22] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. *NeurIPS* (2021).

[23] 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.

[24] 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).

[25] 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).

[26] Chengsong Huang, Qian Liu, Bill Yuchen Lin, Tianyu Pang, Chao Du, and Min Lin. 2023. Lorahub: Efficient cross-task generalization via dynamic lora composition. *arXiv preprint arXiv:2307.13269* (2023).

[27] 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.

[28] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The Stack: 3 TB of permissively licensed source code. *Preprint* (2022).

[29] 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.

[30] 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.

[31] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[32] 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).

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

[34] Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. Towards General Text Embeddings with Multi-stage Contrastive Learning. arXiv:2308.03281 [cs.CL] https://arxiv.org/abs/2308.03281

[35] 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).

[36] 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.

[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] Jiaxing Liu, Chaofeng Sha, and Xin Peng. 2023. An Empirical Study of Parameter-Efficient Fine-Tuning Methods for Pre-Trained Code Models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 397–408.

[39] Shangqing Liu, Yu Chen, Xiaofei Xie, Jingkai Siow, and Yang Liu. 2020. Retrieval-augmented generation for code summarization via hybrid gnn. *arXiv preprint arXiv:2006.05405* (2020).

[40] Shuo Liu, Jacky Keung, Zhen Yang, Fang Liu, Qilin Zhou, and Yihan Liao. 2024. Delving into Parameter-Efficient Fine-Tuning in Code Change Learning: An Empirical Study. *arXiv preprint arXiv:2402.06247* (2024).

[41] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722* (2022).

[42] 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).
[43] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. 2022. PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods. https://github.com/huggingface/peft.
[44] 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).
[45] Sewon Min, Mike Lewis, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2021. Metaicl: Learning to learn in context. *arXiv preprint arXiv:2110.15943* (2021).
[46] 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)*.
[47] Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309* (2023).
[48] 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]
[49] 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.
[50] R OpenAI. 2023. GPT-4 technical report. *arXiv* (2023), 2303–08774.
[51] 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.
[52] Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. *arXiv preprint arXiv:2108.11601* (2021).
[53] 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.
[54] 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.
[55] 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).
[56] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
[57] Iman Saberi, Fatemeh Fard, and Fuxiang Chen. 2024. Utilization of pre-trained language models for adapter-based knowledge transfer in software engineering. *Empirical Software Engineering* 29, 4 (2024), 94.
[58] Iman Saberi and Fatemeh H Fard. 2023. Model-Agnostic Syntactical Information for Pre-Trained Programming Language Models. *arXiv preprint arXiv:2303.06233* (2023).
[59] 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.
[60] Noam Shazeer and Mitchell Stern. 2018. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*. PMLR, 4596–4604.
[61] 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.
[62] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and policy considerations for deep learning in NLP. *arXiv preprint arXiv:1906.02243* (2019).
[63] 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.
[64] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
[65] 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.
[66] 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.
[67] 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).

[68] 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.

[69] 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).

[70] Weishi Wang, Yue Wang, Shafiq Joty, and Steven CH Hoi. 2023. Rap-gen: Retrieval-augmented patch generation with codet5 for automatic program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 146–158.

[71] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[72] 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).

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

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

[75] 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).

[76] 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).

[77] Martin Weyssow, Houari Sahraoui, and Bang Liu. 2022. Better modeling the programming world with code concept graphs-augmented multi-modal learning. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. 21–25.

[78] 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).

[79] 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).

[80] 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.*

[81] 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.

[82] 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

[83] 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).

[84] 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.

[85] 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.

[86] 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

[87] 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.

[88] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating instruction-tuned large language models on code comprehension and generation. *arXiv preprint arXiv:2308.01240* (2023).

[89] 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).

[90] 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.

[91] 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*.

[92] Xin Zhou, DongGyun Han, and David Lo. 2021. Assessing generalizability of codebert. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 425–436.