Resource-Efficient & Effective Code Summarization

Saima Afrin*, Joseph Call†, Khai-Nguyen Nguyen‡, Oscar Chaparro§, Antonio Mastropaolo¶
William & Mary, Department of Computer Science
Williamsburg, Virginia, USA
*safrin@wm.edu, †jbcall@wm.edu, ‡knguyen07@wm.edu, §oscarch@wm.edu, ¶amastropaolo@wm.edu

***Abstract*—Code Language Models (CLMs) have demonstrated high effectiveness in automating software engineering tasks such as bug fixing, code generation, and code documentation. This progress has been driven by the scaling of large models, ranging from millions to trillions of parameters (*e.g.,* GPT-4). However, as models grow in scale, sustainability concerns emerge, as they are extremely resource-intensive, highlighting the need for efficient, environmentally conscious solutions. GreenAI techniques, such as QLoRA (Quantized Low-Rank Adaptation), offer a promising path for dealing with large models' sustainability as they enable resource-efficient model fine-tuning. Previous research has shown the effectiveness of QLoRA in code-related tasks, particularly those involving natural language inputs and code as the target output (NL-to-Code), such as code generation. However, no studies have explored its application to tasks that are fundamentally similar to NL-to-Code (natural language to code) but operate in the opposite direction, such as code summarization. This leaves a gap in understanding how well QLoRA can generalize to Code-to-NL tasks, which are equally important for supporting developers in understanding and maintaining code. To address this gap, we investigate the extent to which QLoRA's capabilities in NL-to-Code tasks can be leveraged and transferred to code summarization, one representative Code-to-NL task. Our study evaluates two state-of-the-art CLMs (CodeLlama and DeepSeek-Coder) across two programming languages: Python and Java. Each model was tasked with generating a meaningful description for Python and Java code methods. The findings of our research confirm previous patterns that emerged when applying QLoRA to source code generation. Notably, we observe that QLoRA not only allows efficient fine-tuning of CLMs for code summarization but also achieves the best results with minimal parameter adjustment compared to full model fine-tuning, which requires expensive recalibration of all model parameters in the traditional fine-tuning process.**

***Index Terms*—Code Summarization, PEFT, Quantization, QLoRA, Code Language Models**

I. Introduction

In recent years, deep learning (DL) generative models, particularly Large Language Models (LLMs) and Code Language Models (CLMs), have transformed key software engineering (SE) activities, including bug fixing, code generation, and code documentation [1–5]. These advances have significantly enhanced automation, driving productivity in SE workflows.

To fully realize their potential, CLMs often require fine-tuning to achieve high accuracy on specific tasks. Prior research [6, 7] has shown that fine-tuned CLMs outperform pre-trained CLMs that rely on in-context learning (ICL), particularly for downstream tasks such as code summarization [8–10]. Fine-tuning allows for deeper calibration of model parameters, resulting in higher adaptability & robustness for specific tasks.

However, fine-tuning large-scale language models—often comprising billions of parameters—demands significant computational resources and time [11]. For instance, training the CodeLlama [12] family of models reportedly required over 1.4 million GPU hours [13], highlighting the substantial effort needed to achieve state-of-the-art performance.

In response to this challenge, researchers have explored sustainable methods to reduce the environmental and computational costs associated with training large-scale models while maintaining high performance [6, 14–16]. Techniques such as model compression and parameter-efficient fine-tuning (PEFT) [6, 15–18] have emerged as promising solutions, enabling efficient training with significantly lower resource demands.

One recent advancement at the intersection of model compression and PEFT is QLoRA (Quantized Low-Rank Adaptation) [19], a technique that combines model size reduction with efficient fine-tuning strategies. QLoRA has been shown to enable cost-effective fine-tuning of CLMs for tasks such as program repair and code generation/completion [6, 20], which fall into the categories of Code-to-Code and NL-to-Code (natural language to code) tasks. These results suggest that QLoRA significantly reduces computational overhead while achieving high effectiveness compared to methods requiring full parameter calibration. Despite these promising results, the applicability of QLoRA to Code-to-NL tasks, such as code summarization, remains unknown. This paper addresses this gap by evaluating QLoRA's effectiveness for code summarization.

Code summarization, like other bi-modal code-related tasks (*e.g.,* code review and code generation), requires reasoning across code and natural language, with the aim to translate complex code logic into accurate, clear, and concise natural language explanations. Given that QLoRA has proven effective for code generation [6], we *hypothesize* that it is equally effective for code summarization. This hypothesis is grounded in the conceptual parallel between teaching a model to generate code and teaching it to summarize code, as both tasks involve an inverse relationship where input and output roles are reversed, with both tasks learning nuanced relationships between natural and programming languages.

To validate this hypothesis, we conducted a systematic evaluation of QLoRA using two state-of-the-art CLMs, CodeLlama [12] and DeepSeek-Coder [21], designed to summarize code methods written in Python and Java from the CodexGLUE's code summarization dataset[1]. We trained these

[1]https://tinyurl.com/axbp8hua

models with QLoRA under varying parameter sizes and compared their performance to full model fine-tuning, analyzing memory usage and predictive accuracy. Additionally, we qualitatively analyzed two statistically significant samples of code methods—one comprising Python methods and the other Java methods—to evaluate how closely the generated summaries align with the ground truth and how effectively they convey equivalent information. This analysis establishes a virtual upper bound on the potential effectiveness of QLoRA for Code-to-NL tasks, particularly code summarization.

Our results show that QLoRA achieves superior predictive performance compared to full fine-tuning while consistently reducing the memory footprint of CLMs. These findings provide compelling evidence of QLoRA's ability to optimize CLMs for resource-intensive, bi-modal code-related tasks, thereby showing its utility across the full spectrum of code-related tasks: Code-to-Code, NL-to-Code, and Code-To-NL.

To the best of our knowledge, this work represents the first large-scale evaluation of QLoRA for code summarization, and it makes the following key contributions:

- A comprehensive analysis of QLoRA's capabilities for code summarization, using two state-of-the-art CLMs across two programming languages, contributing to a broader understanding of resource-efficient training across the full spectrum of code-related tasks.
- Key insights into the trade-offs between memory usage and model performance compared to full model fine-tuning, showcasing QLoRA's ability to achieve remarkable results with substantially reduced resource requirements in the context of code summarization.
- A replication package [22], including data, models, scripts, and documentation, to facilitate reproducibility and further research in this field.

II. Background and Related Work

This section provides the reader with an overview of recent advancements in efficiency-based methods that aim to improve the sustainability of large language models, particularly code language models (CLMs) for code summarization.

A. Code Language Models in Code Summarization

Given the significant potential of SE-related automation through large code models grounded on LLMs, researchers increasingly leveraged these models to support various tasks, including those requiring higher levels of abstraction. One such task is code summarization, which involves working with bi-modal data to translate and summarize code into natural language. In this task, LLMs have proven highly effective [23–27]. CLMs like Codex [28, 29], CodeBERT [30, 31], and T5 [27] excel in understanding code functionality and logic, generating clear and concise summaries. For example, Mastropaolo *et al.* [27] pre-trained a T5-based model on a blend of code and technical natural language before fine-tuning it on various code-related tasks, including code summarization. Their results highlighted the advantages of leveraging transfer learning for bi-modal code-related tasks, particularly code summarization. Haldar *et al.* [32] investigated the use of CodeT5 [33], PaLM2 [34], and Llama2 [35] to generate meaningful code summaries. While CodeT5 was subject to fine-tuning, PaLM2 and Llama2 required no parameter adjustment. The authors' findings reveal that LLMs frequently leverage function names and shared tokens between the code and its summary to optimize predictive performance.

Ahmed *et al.* [9] found that few-shot prompting, which involves providing the model with few examples for generation tasks, significantly improves Codex's performance in code summarization, outperforming smaller pre-trained models like CodeT5. In another study, Sun *et al.* [36] explored CodeLlama [13] and GPT-4 [37] for code summarization and evaluated five prompting techniques (*i.e.,* zero-shot, few-shot, chain-of-thought, critique, and expert). They identified the most effective prompt for guiding GPT-4 to generate in-distribution code summaries.

B. Parameter-Efficient Fine-Tuning and Quantization Methods

Parameter-Efficient Fine-Tuning (PEFT) optimizes fine-tuning by updating only a subset of a model's parameters, rather than the entire model. Common techniques include: (i) Adapters, where additional model layers are introduced to handle a limited set of parameters [38]; (ii) Prompt Tuning, which trains the model to learn from prompts containing task descriptions or canonical examples [39, 40]; and (iii) LoRA (Low-Rank Adaptation), which decomposes weight gradients into low-rank matrices during fine-tuning [41].

PEFT has shown strong performance in tasks such as code generation and summarization, often outperforming fully fine-tuned models. For instance, Wang *et al.* [42] applied Adapter tuning for code search and summarization, while Ayupov *et al.* [15] showcased the effectiveness of Adapters and LoRA in tasks like code summarization and code clone detection. Similarly, Liu *et al.* [43] compared PEFT methods—such as Adapter, LoRA, prefix tuning, and Multi-Head Modification (MHM)—for tasks like defect detection, clone detection, code translation, and code summarization. Recent studies [18, 44] have further explored PEFT techniques in the context of code summarization, highlighting their importance in this domain.

Quantization is a technique for model compression. It aims to reduce the size of a model by preserving only the most essential information encoded in the model's parameters. Specifically, it achieves compression by representing weights or activations in lower-precision formats, such as 8-bit integers, rather than higher-precision formats like 16-bit or 32-bit floats [45, 46]. This approach reduces latency while minimizing any potential loss in accuracy.

In the software engineering domain, the pioneering study by Wei *et al.* [14] represents the first large-scale investigation into the application of quantization techniques for code-related tasks, including code generation and summarization. The authors examined the effects of 8-bit quantization on various code models, such as PLBART [47], CodeT5 [33], InCoder [48], and CodeGen [49]. Their findings revealed that

applying 8-bit quantization to CodeGen and InCoder resulted in improved energy efficiency during code generation, while PLBART and CodeT5 showed similar benefits for code summarization. Notably, these gains in efficiency were achieved with only a minimal reduction in model accuracy.

C. Quantized Low-Rank Adaptation (QLoRA) of CLMs

Dettmers *et al.* [19] recently proposed QLoRA, an approach that combines the LoRA PEFT technique with quantization of LLMs. QLoRA introduces various key innovations, including (i) the 4-bit NormalFloat (NF4) data type, (ii) Double Quantization (DQ), and (iii) a Paged Optimizer. It has been shown to be an efficient fine-tuning method that reduces memory usage while preserving the high performance of LLMs [19]. QLoRA quantizes the pre-trained model's weights to 4-bit precision using NF4, a data type optimized for the normal distribution of neural network weights. Additionally, through double quantization, both the model weights and the quantization constants are quantized, further reducing the memory footprint. To manage memory spikes during gradient checkpointing and prevent out-of-memory errors, QLoRA employs Paged Optimizers. A detailed explanation of QLoRA and the fine-tuning process to achieve its goals is provided in Section III-B.

Limited research has investigated the efficiency of QLoRA for code language models. Yang *et al.* [20] applied QLoRA on models such as CodeLlama [13], StarChat-alpha [50], and Mistral-Instruct-7B [51] to specialize large code models for automatic program repair (APR). Their findings demonstrate that QLoRA effectively supports LLMs in repairing defects in software systems. Weyssow *et al.* [6] compared PEFT techniques to In-Context Learning (ICL) for code generation, concluding that PEFT methods achieved superior results. In addition, the authors also investigated the applicability of QLoRA to CodeLlama 7B, 13B, and 34B Python models, using 8-bit and 4-bit quantization.

While these findings provide valuable insights, a comprehensive evaluation of whether QLoRA can effectively support the entire spectrum of code-related tasks–namely NL-to-Code, Code-to-Code, and Code-to-NL–remains absent. To address this gap, this paper takes a significant first step toward exploring QLoRA's potential across these task categories. Specifically, we focus on Code-to-NL tasks, using code summarization as a representative case study, to evaluate how well QLoRA adapts in scenarios where the model processes code as input and generates natural language as output. This work seeks to deepen the understanding of resource-efficient training methods in software engineering tasks while providing a foundation for future research across diverse bi-modal tasks (*e.g.,* code review automation).

III. Study Methodology

The main goal of this study is to investigate the application QLoRA fine-tuning to code language models (CLMs) for code summarization. QLoRA combines PEFT and quantization techniques, resulting in substantial improvements in memory efficiency during LLM training compared to LoRA [19]. The study addresses the following research question (RQ):

RQ: *How effective and memory-efficient are CLMs for code summarization when fine-tuned with QLoRA, compared to full fine-tuning?*

Through this RQ, we aim to validate our hypothesis that QLoRA is equally effective for code summarization as it is for code generation [6].

To answer the RQ, we examine two state-of-the-art code models: CodeLlama [12] and DeepSeekCoder [21]. Each model is trained and evaluated on the CodexGLUE code summarization benchmark [52], particularly the dataset comprising Python and Java code methods and their respective summaries [53].

Additionally, we investigate the impact of scaling up the parameters of CLMs during QLoRA-based training, measuring changes in GPU memory usage and overall predictive performance for code summarization. This analysis aims to determine whether larger models retain their performance advantage, as demonstrated in previous studies [23–25, 54], where increasing the number of model parameters has consistently improved task-specific performance.

We also investigate the generalizability of QLoRA for LLMs that, while widely utilized for automating SE-related tasks, were not primarily designed for such tasks. The details of this analysis are provided in Section IV-C.

A. Code Language Models (CLMs)

For our study, we selected two families of state-of-the-art CLMs: CodeLlama [13] and DeepSeekCoder [21]. The models have been frequently investigated in prior work [21, 55, 56].

Our selection includes models with distinct architectural or training features, making them well-suited for code summarization. For example, the models are available in both instruction-tuned and non-instruction-tuned variants. Instruction-tuned models are optimized to process human-like instructions, making them particularly effective at manipulating natural language and code. This additional capability can be harnessed even in the context of QLoRA training, as demonstrated in prior work [57, 58].

CodeLlama [12] is a family of open-source LLMs tailored for coding tasks. It is based on the general-purpose Llama-2 model [59], with further training on a corpus of 500B tokens that include both natural language and code. CodeLlama is available in several variants [60], each designed for specific use cases: a general-purpose coding model, an Instruct variant optimized for instruction tuning, and a Python-specialized version. The model sizes range from 7B to 70B parameters, and all versions are publicly accessible. CodeLlama has demonstrated strong performance in automating a range of code-related tasks [61, 62], making it a representative model for our study. We used the general-purpose *Instruct* version featuring 7B and 34B parameters in our experiments.

DeepSeek-Coder [21] is a set of open-source LLMs ranging from 1B to 33B parameters. These models are offered in two

configurations: Instruct, optimized for instruction tuning, and Base. Trained on a dataset of two trillion tokens, including code-specific data, DeepSeek-Coder has been shown capable of outperforming larger models such as GPT-3.5 [63], while the small-sized version featuring 6.7B parameters has proven highly competitive to CodeLlama's 33B variant. For this study, we used the *Instruct* version of DeepSeek-Coder in three variants, 1.3B, 6.7B, and 33B parameters. This selection served two different goals: (i) it allowed us to compare the performance of QLoRA-optimized models against fully fine-tuned models by contrasting the results achieved by DeepSeek-Coder 1.3B in both configurations; and (ii) it enabled a comparison between small-sized (CodeLlama 7B *vs.* DeepSeek-Coder 6.7B) and mid-sized models (*e.g.,* CodeLlama 34B *vs*. DeepSeek-Coder 33B), providing insights into how QLoRA fine-tuning impacts performance across different model sizes.

B. The QLoRA Fine-tuning Technique

QLoRA employs two innovative techniques for effective 4-bit finetuning: 4-bit NF4 quantization and Double Quantization, along with Paged Optimizers to manage memory efficiently during gradient checkpointing.

1) NF4 Quantization: The core of QLoRA's approach lies in a method designed to efficiently quantize neural network weights into a 4-bit format which, uses NF4, a novel data type designed for AI applications. The 4-bit Normal Float (NF4) data type is based on Quantile Quantization [64], which ensures an even distribution of tensor values across quantization bins or categories. Using fast quantile approximation algorithms, QLoRA can estimate quantiles without the high computational costs associated with precise quantile calculations.

During this process, the neural network weights, which generally follow a zero-centered normal distribution, are adjusted to fit a predefined range. This normalization aligns the weight tensors with the range of the data type, allowing for more effective quantization by matching the tensor's value distribution to that of the quantized format.

2) Double Quantization: To further reduce memory footprint, QLoRA follows a two-step approach: (i) the model weights are quantized to 4-bit precision using NF4, and (ii) the quantization constants (scales and zero-points) from the first step are quantized to a lower precision. QLoRA implements Blockwise k-bit Quantization, where weights are divided into distinct blocks that are independently quantized, rather than quantizing all weights collectively. This method generates multiple quantization constants, which can undergo a second round of quantization, providing additional memory savings.

3) Paged Optimizer: When training large models, gradient checkpointing comes in handy as a technique to reduce memory usage during model training, yet memory spikes can still occur when processing mini-batch with a long sequence of input tokens. Paged optimizers minimize GPU memory use by storing states in CPU memory and transferring them as needed.

Fig. 1 depicts the fine-tuning process of QLoRA, an extension of LoRA that, as noted, utilizes NF4 for efficient weight storage and BFloat16 for computations and gradient calculations. The addition of paged optimizer memory management further enhances efficiency, making QLoRA particularly suitable for resource-constrained environments.

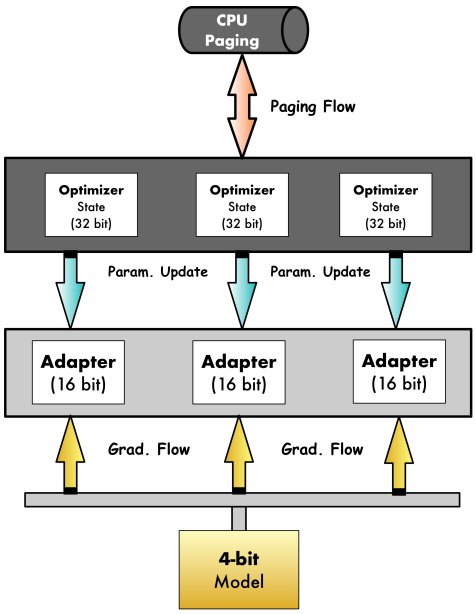

Fig. 1. QLoRA finetuning with paged optimizers [19]

In our study, we selected the QLoRA parameter configuration outlined in Table I, which includes three parameters: (i) `lora_r`, (ii) `lora_alpha`, and (iii) `lora_dropout`. These were kept constant for each QLoRA fine-tuning instance. The choice of the hyperparameter values follows established best practices to ensure precision while minimizing resource consumption [19, 41].

TABLE I
QLoRA HYPERPARAMETERS USED IN OUR EXPERIMENTS

Parameter	Description	Value
`lora_r`	*lora attention dimension/ rank*	8
`lora_alpha`	*lora scaling parameter*	16
`lora_dropout`	*lora dropout probability*	0.1

C. Dataset and Model Training

We employed the *Code-to-Text* dataset from the CodeXGLUE benchmark [52, 53] to train and evaluate all QLoRA-optimized models, focusing specifically on Java and Python. The benchmark consists of pairs of code methods and their associated natural language descriptions, extracted from ~6 million instances of human-written code documentation.

Our decision to leverage CodeXGLUE was driven by its extensive use in prior research for studying LLMs in code-related tasks [24, 27, 65–67].

Table II presents a summary of the datasets employed for training and evaluation. To this extent, we train each QLoRA-

optimized model using a fixed set of hyperparameters, as detailed in Table I. Each model was trained for *10 epochs* with a consistent *batch size of 32* maintained throughout all experiments. To prevent overfitting, we implemented an early stopping, saving a new checkpoint after every 5,000 training steps, and monitoring the performance of the models using the METEOR score, which acts as a highly reliable proxy for differences exceeding 2 points in evaluating the quality of code summaries as perceived by humans [68]. In particular, the training process stops if no improvements in the METEOR score are observed after 15K steps, which equals to a window of 3. This approach allowed us to effectively monitor model performance and ensured that we retained the best-performing checkpoint of the models for both programming languages.

TABLE II
#NUMBER OF DATA INSTANCES IN TRAINING, VALIDATION, AND TESTING SPLITS

Language	Training	Validation	Testing
Java	164,923	5,183	10,955
Python	251,820	13,914	14,918

During training, each model takes as input tokens the tokenized code from the `code_tokens` field in the JSON file, corresponding to either Java[2] or Python[3]. The output is the sequence of natural language tokens provided in the `docstring_tokens` field and joined together to form a string, specific to each programming language. We configured the maximum sequence length to 300 tokens during the training stage based on our analysis of the token distribution for code and natural language in the Code-to-Text dataset.

In our implementation of QLoRA, we followed the findings from [19] and applied QLoRA to all linear layers of the networks (*e.g.,* Feed-Forward Layers, Self-Attention Layers, and Projection Layers). This approach enables QLoRA to effectively adapt to the task at hand by leveraging the parameter space considered essential for optimal performance.

The training process for full fine-tuning followed the same configuration as used for QLoRA-optimized models, including batch size, number of training epochs, and early stopping.

D. Metrics and Experimental Procedure

We started by fine-tuning DeepSeek-Coder 1.3B using QLoRA with the dataset described in Section III-C. As outlined in Section III-A, we limited full fine-tuning (FFT) to the smaller model variants included in our study to mitigate the substantial computational costs associated with adjusting all parameters of larger models (*e.g.,* DeepSeek-Coder 33B).

We proceeded by evaluating the performance of the model when generating code summaries for Java and Python methods. To this end, we relied on metrics that have been widely used in prior code summarization research [5, 69, 70]:

[2]https://zenodo.org/record/7857872/files/java.zip

[3]https://zenodo.org/record/7857872/files/python.zip

BLEU (BilinguaL Evaluation Understudy) [71] measures the similarity between candidate (predicted) summaries and reference (oracle) summaries. This metric assesses the overlap of n-grams within the two summaries, ranging from 0 (completely dissimilar summaries) to 1 (identical summaries). We compute the BLEU score at the sentence level, fixing $n = 4$.

METEOR (Metric for Evaluation of Translation with Explicit ORdering) [72] is computed as the harmonic mean of unigram precision and recall, with recall given a higher weight than precision. Unlike BLEU, METEOR utilizes stemming and synonym matching to align more closely with human judgments of sentence similarity. METEOR ranges from 0 to 1, with a value of 1 indicating two identical sentences.

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [73] consists of a set of metrics for evaluating both automatic text summarization and machine translation methods. ROUGE metrics compare automatically generated summaries or translations against a set of reference summaries, typically authored by humans. In line with Roy *et al.* [68], we computed ROUGE-N(1-4), ROUGE-L, and ROUGE-W. ROUGE-N measures the number of matching n-grams between the generated summary and the reference summary with results reported in terms of recall, precision, and F1-score.

chrF (character n-gram F-score) [74] measures the similarity between generated and reference summaries at the character level (rather than at the token level, done by the above metrics), reporting a F1-score value.

BERTScore [75] computes sentence similarity using the embedding of a BERT model [76] trained on English textual data. We report the F1-score (BERTScore-F1).

SIDE (Summary Alignment to Code Semantics) [3] offers an automated method for evaluating the alignment between a Java method and its corresponding summary. It produces a score within the range of [-1, 1], where values closer to 1 indicate a stronger alignment between the comment and the documented code component. Conversely, lower SIDE scores signify weaker alignment, highlighting discrepancies between the summary and the code.

Next, we conducted a comprehensive end-to-end, task-specific fine-tuning of DeepSeek-Coder, updating 1.3 billion parameters. This process, backpropagates gradients through the entire model, enabling optimal adjustment of each parameter to enhance task-specific performance (*i.e.,* code summarization).

Finally, we assessed whether there are statistically significant differences in performance between fully fine-tuned models and QLoRA-optimized models. To this extent, we employed the Wilcoxon signed-rank test [77], and measured the effect size using Cliff's Delta (d) [78]. The effect sizes are categorized as follows: negligible if $|d| < 0.10$, small if $0.10 \leq |d| < 0.33$, medium if $0.33 \leq |d| < 0.474$, and large if $|d| \geq 0.474$. We used a 95% significance level across all tests and, since we tested our hypotheses through multiple tests, we adjusted the p-values using Holm's correction

procedure [79]. The tests were computed for every metric included in our evaluation.

To investigate the impact of various sizes of models, we fine-tuned four model variants with QLoRA optimization: CodeLlama 7B/34B and DeepSeek-Coder 6.7B/33B. Next, we evaluated the performance of each model configuration in generating meaningful code descriptions for Python and Java methods. Additionally, we applied the Wilcoxon signed-rank test to analyze performance differences in evaluation metrics across models of varying sizes.

Each experiment was performed on a server running Ubuntu 22.04.5 LTS (GNU/Linux 5.15.0-125-generic x86_64), equipped with two Nvidia L40S GPUs, each featuring 48GB of graphics memory.

E. Qualitative Analysis

Evaluating code summarization can present unique challenges, particularly in dealing with semantically equivalent summaries. For example, in Fig. 2, the two summaries indicated with $S1$ and $S2$, describe the same functionality of a code snippet but use different phrasing. This discrepancy poses a challenge because traditional evaluation metrics like BLEU or ROUGE rely heavily on exact word matches and may penalize the generated summary $S1$ for not being identical to the ground truth $S2$. In addition, Mastropaolo *et al.* [3] have recently demonstrated that word-overlap metrics like BLEU, and even embedding-based metrics such as BERTScore [75], can only capture one of the several dimensions pertaining to the evaluation of code summarizers. Thus, to provide a more accurate assessment of the capabilities of QLoRA in fine-tuning models for bi-modal software engineering tasks, we manually analyzed two statistically significant, randomly selected samples: one consisting of 384 Java methods and the other of 384 Python methods, generated by the best-performing QLoRA-optimized model identified in our study.

With this manual analysis, we aimed to better understand the nuances in code summarization tasks that automated metrics might miss, offering a comprehensive evaluation of how effectively QLoRA-optimized models can support bi-modal SE-related tasks across two programming languages.

For this analysis, two paper authors independently reviewed the 768 incorrectly generated summaries, equally split between Python and Java (*i.e.,* 384 + 384). Any conflicts were resolved through open discussion between the reviewers, involving a third author when needed. The summaries were sampled from the set of incorrect predictions made by CodeLlama-34B, the best-performing model. Each prediction was classified as:

- *Semantically equivalent*: The ground truth and the model's prediction use different wording but convey exactly the same information to the developer.
- *Meaningful code description*: These cases represent instances where the model generated a code summary that not only conveyed the intended information but was of better quality than the ground truth. A few examples are reported in Fig. 3.

```
public String read() throws IOException {
    Closer closer = Closer.create();
    try {
         Reader reader = closer.register(openStream());
         return CharStreams.toString(reader);
    } catch ( Throwable e ) {
         throw closer.rethrow (e);
    } finally { closer.close(); }
}
```

S1: *Gets the textual information from this source and represent it as string.*

S2: *Reads the contents of this source as a string.*

Fig. 2. Semantically equivalent Java code summaries.

- *Partially equivalent*: The prediction includes only part of the information conveyed in the ground truth. While these predictions can still be useful for the developer, some code adjustments are needed to align the prediction with the original method.
- *Incorrect*: The code summary predicted by the model is documenting something else and not the underlying code.

To assess inter-rater reliability among the evaluators, we calculated Krippendorff's α coefficient [80]. For the analysis of Java elements, the α coefficient was 0.752, and for Python, it was 0.803. In both cases, the α that ranges between [-1;1] indicated a high level of agreement between the two evaluators. Following this, we report the percentage of instances in each category detailed above.

IV. Results and Discussion

In this section, we present and discuss the results of our study addressing our RQ, which aims to evaluate the effectiveness and memory footprint of the CLMs when fine-tuned with QLoRA, compared to full fine-tuning.

A. QLoRA vs Full Fine-Tuning

As described in Section III-D, we fine-tuned DeepSeek-Coder 1.3B under two configurations: QLoRA fine-tuning and full fine-tuning (FFT). The CodeLlama models are only fine-tuned with QLoRA due to the excessive computational costs associated with full fine-tuning, which we could not afford. Table IV illustrates the numbers of *total model parameters* and *trainable parameters* for our selected models.

Table III summarizes the performance results for the five models optimized using QLoRA, alongside the Full Fine-Tuning (FFT) of DeepSeek-Coder 1.3B. A key insight that emerges after observing the second and third rows is that QLoRA-based fine-tuning consistently delivers superior performance compared to full fine-tuning of DeepSeek-Coder 1.3B across the two programming languages. For example, in terms of the METEOR score, the QLoRA fine-tuned model surpasses its fully fine-tuned counterpart by approximately 2% for both Python and Java. Similarly, based on the ROUGE-L metric, performance improvements range from 1.9% to 2.7%, with the largest gains observed for Java. For each metric reported in Table III, the observed differences were found to be statistically significant (based on a Wilcoxon signed-rank

TABLE III
PERFORMANCE OF DIFFERENT QLORA FINE-TUNED PARAMETER SIZE MODELS. THE BEST RESULTS ACROSS ALL THE EXPERIMENTED CONFIGURATIONS ARE HIGHLIGHTED IN YELLOW.

Model	Parameter Size	Training	Dataset Type	BLEU	METEOR	Rouge-L	chrF	BERTScore F1	SIDE
CodeLlama	7B	QLoRA	Python	8.9%	33.9%	**36.5%**	29.7%	88.8%	-
			Java	10.3%	35.8%	**37.4%**	30.9%	89.0%	87.5%
	34B	QLoRA	Python	**10.7%**	**35.3%**	35.2%	**31.3%**	89.0%	-
			Java	**11.6%**	**37.2%**	36.7%	**32.0%**	**89.1%**	86.6%
DeepSeek-Coder	1.3B	**FFT**	Python	6.5%	32.3%	30.1%	27.4%	88.2%	-
			Java	8.2%	33.4%	32.2%	28.8%	88.4%	87.3%
		QLoRA	Python	7.4%	34.2%	32.0%	28.0%	88.5%	-
			Java	8.7%	35.1%	33.9%	29.0%	88.6%	**88.1%**
	6.7B	QLoRA	Python	8.6%	36.5%	33.9%	29.4%	88.8%	-
			Java	9.9%	37.3%	35.5%	30.1%	89.0%	87.9%
	33B	QLoRA	Python	10.5%	37.8%	35.2%	31.2%	89.0%	-
			Java	10.9%	38.1%	36.4%	31.5%	89.0%	87.4%

TABLE IV
SUMMARY OF MODEL TRAINING PARAMETERS AND MEMORY UTILIZATION IN MEGABYTES (MB)

Training Strategy	Model	Peak GPU Mem. Consumption (MB)	Trainable Parameters	Model's Parameters	Trainable %
QLoRA Fine-Tuning	CodeLlama-7b	11,877	20,277,376	6,758,820,064	0.300
	CodeLlama-34b	37,424	54,781,952	33,798,752,256	0.162
	DeepSeek-Coder-1.3b	5,154	7,770,112	1,354,242,048	0.574
	DeepSeek-Coder-6.7b	12,894	20,279,296	6,760,792,064	0.300
	DeepSeek-Coder-33b	39,724	61,898,752	33,404,890,112	0.185
Full Fine-Tuning	DeepSeek-Coder-1.3b	16,776	1,354,242,048	1,354,242,048	100

test at 95% significance), though the effect size is negligible. The detailed results of this analysis are available in our online replication package [22].

Table IV shows the results of our analysis of the peak GPU memory consumption during model fine-tuning. The table reveals a substantial difference between QLoRA and full fine-tuning. FFT of the 1.3B parameters requires an average of approximately 16GB of GPU memory, while QLoRA fine-tuning significantly reduces memory usage, requiring only 5GB on average–saving 10GB compared to FFT (*i.e.,* about a third of the FFT memory footprint). This reduction in memory consumption can be attributed to the considerably lower number of trainable parameters in QLoRA (Table IV–Columns 4 and 5), which represents a significant scale difference (millions vs. billions) compared to training all parameters. The freed-up memory can be used to support additional tasks, enable larger batch sizes, or improve overall computational efficiency, making QLoRA a more practical and scalable solution for resource-constrained environments.

The results indicate that QLoRA is not only effective in optimizing resource efficiency but also in outperforming full fine-tuning. These findings align with those presented in the pioneering work that introduced QLoRA [19].

In the software engineering literature, Weyssow *et al.* [6] demonstrated that for coding activities–particularly code generation–the use of LoRA adapters for fine-tuning large language models outperforms FFT. Furthermore, in the same study, they showed that QLoRA fine-tuning surpasses LoRA fine-tuning for the same task, establishing a clear hierarchy of fine-tuning strategies: FFT < LoRA fine-tuning < QLoRA fine-tuning. Hence, if LoRA fine-tuning already outperforms FFT, this suggests that once the underlying model's knowledge, distilled within its parameters, is frozen–as in LoRA and QLoRA fine-tuning–adapting a smaller subset of parameters is sufficient to effectively capture the nuances of the intended task. As we show, QLoRA improves model performance while reducing memory footprint. This result might seem counter-intuitive, given that QLoRA relies on extreme quantization to optimize memory usage. However, a possible explanation for this behavior comes from Dettmers *et al.* (the authors of QLoRA), who observe that any performance degradation due to information loss during quantization is not only fully recovered but often surpassed through the fine-tuning of LoRA modules after the quantization process.

> ***Finding*$_1$:** QLoRA fine-tuning for code summarization delivers performance on par with what is observed for other software engineering tasks such as code generation [6]. By optimizing a limited subset of quantized parameters, it outperforms full fine-tuning in terms of predictive performance and memory consumption.

Examining the impact of varying parameter counts, the results align with our expectations: larger models consistently outperform their smaller counterparts when QLoRA is applied to support code summarization tasks (see Table III). For example, CodeLlama-34B demonstrates significantly higher

performance compared to its 7-billion-parameter variant, and a similar pattern is found with DeepSeek-Coder, where larger versions achieve superior results. These trends hold true across both programming languages, underscoring the effect of QLoRA regardless of the model size. However, this improvement comes at a cost. Larger models demand significantly more GPU memory during training, with usage peaking at 40 GB when fine-tuning DeepSeek-Coder 33B and 37.5 GB when fine-tuning CodeLlama 34B, as detailed in Table IV. This highlights the trade-off between model size, performance, and resource requirements. Depending on the final application and available hardware, one may need to prioritize performance over resource consumption or vice versa. In other words, if GPU memory allocation is limited, sacrificing some performance may be a reasonable trade-off, especially considering the capabilities of models with around 7B parameters. The observed improvements resulted in statistically significant differences, though the effect sizes of these differences are negligible. The detailed results of all the statistical analyses are available in our online replication package [22]. Scaling up to DeepSeek-Coder 33B, the improvements, while still statistically significant, exhibit diminishing returns. The performance gap narrows across all evaluated metrics, as evidenced by the negligible effect sizes. Although larger models generally offer greater capacity, the diminishing improvements suggest that further scaling might not always justify the increased computational resources, particularly for code summarization.

In contrast, for CodeLlama, scaling from 7B to 34B yields more pronounced gains. The larger 34B variant achieves a 3-4% improvement in METEOR scores over its smaller counterpart, equating to an 11.8% improvement for Python and 8.4% for Java. These results highlight the effectiveness of both the CodeLlama model family and DeepSeek-Coder in leveraging increased parameter counts to enhance performance. This trend becomes particularly apparent when examining the top-performing models in Table III, where the best results across all experimental configurations are highlighted in yellow. Notably, four out of six metrics in our evaluation reach their highest values with models from the CodeLlama family that have a parameter count exceeding 30B. The performance differences remain consistent across embedding-based metrics (BERTScore-F1 for both languages, SIDE for Java) and are statistically significant but with negligible effect sizes, suggesting limited practical impact. This reinforces that while larger models can improve performance, the gains may not justify the increased GPU memory consumption.

Finding$_2$**:** Larger models generally offer greater capacity and potential better support for code summarization, but they eventually reach a point of diminishing return. However, if maximizing performance is the primaryobjective, CodeLlama 34B delivers the best outcomes for code summarization, with improvements of 11.8% for Python and 8.4% for Java.

B. Qualitative Analysis

Fig. 3 presents four triplets of ⟨Method, Target$_{summary}$ or GT, Predicted$_{summary}$ or PR⟩. The examples are divided by programming language, with two for each language. The top section features Java examples, while the bottom showcases two Python examples. In all cases, the model-generated comments are not only accurate compared to the ground truth but also offer improvements over it. We remind the reader that this type of code comment falls under the category of *meaningful code descriptions* (Section III-E).

Specifically, focusing on the first triplet ❶, the developer-provided ground truth summary, `attempt to exit from an already switched user`, encapsulates the method's basic functionality. However, the CodeLlama 34B model optimized using QLoRA, generates a summary that clarifies the method's logic, explicitly documenting that the method `attempts to exit the current user by returning the original user that was being impersonated`. This provides valuable additional information, specifically noting that `the method returns the original user who was being impersonated`. This distinction makes the summary more comprehensive, as it clarifies the intended logic and provides insights into the rationale of the code.

In ❷, the predicted summary demonstrates significant improvement, as the model captures details that the developer overlooked or deemed unnecessary, such as the process of extracting tokens. The model not only identifies these elements but also elaborates on them, providing a more comprehensive and actionable summary: `extracts the scope from the access token and converts them to grant authorities`. This enhanced prediction demonstrates the model's capability to infer additional context and generate summaries that not only "copy" tokens already present in the method but also synthesize new information. For instance, the word-token `access` was inferred through the model's deeper understanding of the code's logic and intent. This ability allows for the creation of more comprehensive and insightful summaries.

Turning to the Python first example ❸, the prediction adds depth by conveying an additional message: that the `dagrun must be retrieved based on the most recent execution date`. This enriched context provides developers with summaries that are not only concise but also contextually informative. The second Python example (❹) illustrates that, despite the significantly smaller number of parameters adjusted during fine-tuning (*i.e.,* millions rather than billions), the performance of the QLoRA-optimized model remains, even for tasks demanding high contextual reasoning. Notably, the model identifies an important detail revealed only at the conclusion of the method: *list_py_paths is a recursive method*. This ability to detect nuanced information demonstrates the model's effectiveness in generating meaningful and context-aware summaries despite the limited parameter adjustment, quantized and dequantized, as explained in Section III-B.

1

```java
protected Authentication attemptExitUser(HttpServletRequest
request)
        throws AuthenticationCredentialsNotFoundException {
    Authentication current =
          SecurityContextHolder.getContext().getAuthentication();
    if (null == current) {
        throw new AuthenticationCredentialsNotFoundException(
            this.messages.getMessage(
                 "SwitchUserFilter.noCurrentUser",
                 "No current user associated with this
                  request"));
    }
    Authentication original = getSourceAuthentication(current);
    if (original == null) {
        this.logger.debug("Could not find original user
                                Authentication object!");
        throw new AuthenticationCredentialsNotFoundException(
           this.messages.getMessage(
                "SwitchUserFilter.noOriginalAuthentication",
               "Could not find original Authentication
                object"));
    }
    … …

    return original;
}
```

GT: ***attempt to exit from an already switched user***

PR: ***attempt to exit the current user by returning the original user that was being impersonated.***

2

```java
@Override
public Collection<GrantedAuthority> convert(Jwt jwt) {
      return getScopes(jwt).stream()
            .map(authority -> SCOPE_AUTHORITY_PREFIX + authority)
            .map(SimpleGrantedAuthority::new)
            .collect(Collectors.toList());
}
```

GT: ***extract the authorities***

PR: ***extracts the scopes from the access token and converts them to grantedauthorities.***

3

```python
def get_previous_dagrun(self, session=None):
        return session.query(DagRun).filter(
                  DagRun.dag_id == self.dag_id,
                  DagRun.execution_date < self.execution_date
        ).order_by(
            DagRun.execution_date.desc()
        ).first()
```

GT: ***the previous dagrun if there is one***

PR: ***returns the dagrun for the most recent execution_date***

4

```python
def list_py_file_paths(directory, safe_mode=True,
                       include_examples=None):
    …
    …
    for root, dirs, files in os.walk(directory,followlinks=True):
        patterns = patterns_by_dir.get(root, [])
            ignore_file = os.path.join(root, '.airflowignore')
        …
        …
    if include_examples:
        import airflow.example_dags
        example_dag_folder = airflow.example_dags.__path__[0]
        file_paths.extend(list_py_file_paths(example_dag_folder,
                          safe_mode, False))
    return file_paths
```

GT: ***traverse a directory and look for python files.***

PR: ***get all file paths in a directory recursively that end in py.***

Fig. 3. Examples of predictions made by CodeLlama 34B that have been labeled as *meaningful code summaries*.

These instances align with the findings of the manual investigation conducted on 384 incorrect Java summaries and 384 incorrect Python summaries.

For Java, 15.36% of conflicts arose during the labeling process. These conflicts, resolved by a third author who was not involved in the initial labeling, resulted in the following breakdown: 31.07% of summaries were deemed semantically equivalent to the ground truth, 53.0% were partially equivalent, and in 8.87% of cases, CodeLlama 34B provided summaries that were more accurate and informative than those written by developers. Finally, 7.57% of the summaries were classified as incorrect.

A similar pattern was observed for Python, with slight variations in the distribution across categories. Specifically, CodeLlama 34B generated 37.86% of summaries as a real developer would do (*i.e.,* semantically equivalent summaries), 53.0% were partially equivalent, and in 3.66% of cases, CodeLlama 34B produced recommendations superior to those of developers. Finally, 5.48% of the recommendations were found to be incorrect.

C. Can We Translate the QLoRA Benefits for Code Summarization to General-Purpose Language Models?

Our investigation demonstrated that QLoRA can serve as a resource-efficient training strategy for code summarization, paving the way for advancements in Code-to-NL tasks. However, an open question remains: *can these findings be generalized to models that have been pre-trained—not solely but partially—on software engineering data?*

The rationale behind this question lies in the growing adoption of hybrid models, such as Phi-3 mini [81], which have been benchmarked extensively on coding tasks while also serving as baselines in comparisons against both code-specific and general-purpose language models [82]. These models blur the lines between domain-specific and general-purpose architectures, offering a unique ground for evaluating the transferability of QLoRA's fine-tuning benefits.

For such analysis, we selected Phi-3 mini [81], a 3.8-billion-parameter model introduced by Microsoft in 2024 that was shown to achieve results on par with Llama 3 [83], Meta's state-of-the-art 7-billion-parameter model, when applied to code-related tasks such as code generation. Phi-3 mini is nearly half the size of Llama 3 [83]. We chose this Microsoft model for the generalizability analysis due to its popularity and its frequent use as a baseline for coding tasks [82, 84].

We trained Phi-3 mini using two distinct configurations, replicating the approach used for DeepSeek-Coder 1.3B (Section III). Specifically, we conducted both full model fine-tuning and QLoRA fine-tuning, followed by model evaluation in each scenario. The training and evaluation processes were carried out as outlined in Section III.

Table V presents the results of our experiments with Phi-3 mini. Notably, focusing on the BLEU metrics, the QLoRA-optimized model consistently outperforms the fully fine-tuned model, showing a performance improvement of 2% for Python and 0.9% for Java. This trend extends to other metrics, such as METEOR, ROUGE-L, and chrF, where the QLoRA-optimized model surpasses the FFT model by a margin of approximately 2–3%. Further analysis compares the performance of code models with the general-purpose phi-3 mini model. While code models demonstrate better performance than Phi-3 mini, the observed gap is not substantial. To validate this finding, we conducted a Wilcoxon signed-rank test between the Phi-3

TABLE V
PERFORMANCE OF PHI-3-MINI WHEN FULLY FINE-TUNED AND QLORA FINE-TUNED

Model	Parameter Size	Training	Dataset Type	BLEU	METEOR	Rouge-L	chrF	BERTScore F1	SIDE
Phi-3-mini	3.8B	**FFT**	Python	6.1%	31.8%	29.4%	26.3%	88.1%	-
			Java	7.9%	33.6%	32.1%	27.7%	88.5%	87.3%
		QLoRA	Python	7.9%	35.4%	32.9%	28.5%	88.7%	-
			Java	8.7%	35.9%	34.4%	28.9%	88.8%	88.1%

mini-3.8B and DeepSeek-Coder-1.3B models, evaluating both fully fine-tuned and QLoRA fine-tuned versions. The analysis failed to reveal statistically significant differences for Java (regardless of the evaluation metric), whereas, for Python, this behavior was observed only for the BLEU metric in the fully fine-tuned setup. These analysis details are included in our replication package [22].

V. IMPLICATIONS OF OUR FINDINGS

Among the various applications of code-related bi-modal tasks, code summarization stands as a fundamental endeavor in software engineering, playing a crucial role in enhancing developer productivity [2, 55], improving code comprehension [5, 25], and supporting software maintenance [20].

Based on our findings, we draw the following implications:

Improving CLMs Sustainability via QLoRA fine-tuning. By applying QLoRA fine-tuning to two SoTA code models for code summarization activities, we demonstrated that it achieves competitive results compared to full model fine-tuning. Additionally, QLoRA significantly reduces memory requirements, cutting the memory footprint by approximately a third (see Table IV), which enhances the scalability, sustainability, and usability of advanced AI systems built on large language models for code.

QLoRA streamlines CLMs training for code summarization with comparable success to code generation. The successful adaptation of QLoRA for code summarization underscores its potential to enhance a wide range of code-related tasks through efficient fine-tuning. In software engineering research, this paves the way for exploring QLoRA's capability to fine-tune CLMs for hybrid tasks that integrate code and natural language, such as code review. Examining how QLoRA performs in these hybrid scenarios could unlock new possibilities for leveraging code models in complex, real-world applications.

VI. THREATS TO VALIDITY

Construct Validity threats pertain to the relationship between theory and observation, primarily concerning the measurements used to answer our research questions. In this context, the main threat in our study lies in the selection of metrics to evaluate the quality of the generated summaries. As outlined in Section III-D, we utilized well-established metrics for assessing code summary quality [85], along with newer metrics such as SIDE [3].

Internal Validity threats relate to factors internal to our study that could affect the achieved results. One possible threat can be the selected models to conduct the analysis. We partially mitigate this issue by considering models from the state-of-the-art that have been used in several prior studies [86–89].

Another threat is the selection of the QLoRA hyperparameters. In this regard, we resorted to the literature and particularly followed the recommendations provided in the original paper that introduced QLoRA [19].

A further potential limitation concerns memory measurement, which was conducted based on a single run within a specific environment. Factors such as background processes or system load variations could impact memory usage metrics. To mitigate this, we maintained a controlled and consistent environment across experiments. However, we acknowledge that conducting multiple runs under varying conditions could offer additional insights.

Conclusion Validity threats involve the link between the experimental process and its outcomes. As detailed in Section III-D, where appropriate, our conclusions are supported by suitable statistical methods.

External Validity threats concern the generalizability of our findings. In this regard, our study focuses solely on code summarization as a representative bi-modal task. Furthermore, although we conducted experiments using Python and Java—widely studied programming languages in software engineering automation [11, 12, 21]—we recognize that results may differ for other programming languages.

VII. CONCLUSIONS AND FUTURE WORK

This study found that QLoRA is as effective for code summarization as it is for code generation—a task similar in intent but distinct in execution. Specifically, our findings show that QLoRA not only matches but consistently outperforms full model fine-tuning, delivering superior results while significantly improving resource efficiency. Notably, QLoRA excels in memory utilization, positioning it as a practical solution for resource-constrained environments.

Building on the positive impact of QLoRA fine-tuning for complex code-related tasks, our future research will focus on achieving a balance between efficiency and performance when deploying QLoRA in practical applications, such as live coding assistance tools.

REFERENCES

[1] Y. Charalambous, N. Tihanyi, R. Jain, Y. Sun, M. A. Ferrag, and L. C. Cordeiro, "A new era in software security: Towards self-healing software

via large language models and formal verification," *arXiv preprint arXiv:2305.14752*, 2023.

[2] A. Mastropaolo, S. Scalabrino, N. Cooper, D. N. Palacio, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Studying the usage of text-to-text transfer transformer to support code-related tasks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering*. IEEE, 2021, pp. 336–347.

[3] A. Mastropaolo, M. Ciniselli, M. Di Penta, and G. Bavota, "Evaluating code summarization techniques: A new metric and an empirical characterization," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[4] H. Tian, W. Lu, T. O. Li, X. Tang, S.-C. Cheung, J. Klein, and T. F. Bissyandé, "Is chatgpt the ultimate programming assistant–how far is it?" *arXiv preprint arXiv:2304.11938*, 2023.

[5] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1385–1397.

[6] M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui, "Exploring parameter-efficient fine-tuning techniques for code generation with large language models," *arXiv preprint arXiv:2308.10462*, 2023.

[7] H. Liu, D. Tam, M. Muqeeth, J. Mohta, T. Huang, M. Bansal, and C. A. Raffel, "Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning," *Advances in Neural Information Processing Systems*, vol. 35, pp. 1950–1965, 2022.

[8] T. Ahmed, K. S. Pai, P. Devanbu, and E. Barr, "Automatic semantic augmentation of language model prompts (for code summarization)," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[9] T. Ahmed and P. Devanbu, "Few-shot training llms for project-specific code-summarization," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[10] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence," in *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering*, 2022, pp. 382–394.

[11] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, 2023.

[12] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[13] "Codellama," https://github.com/meta-llama/codellama/tree/main.

[14] X. Wei, S. K. Gonugondla, S. Wang, W. Ahmad, B. Ray, H. Qian, X. Li, V. Kumar, Z. Wang, Y. Tian *et al.*, "Towards greener yet powerful code generation via quantization: An empirical study," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 224–236.

[15] S. Ayupov and N. Chirkova, "Parameter-efficient finetuning of transformers for source code," *arXiv preprint arXiv:2212.05901*, 2022.

[16] J. Lu, L. Yu, X. Li, L. Yang, and C. Zuo, "Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning," in *2023 IEEE 34th International Symposium on Software Reliability Engineering*. IEEE, 2023, pp. 647–658.

[17] C.-Y. Su and C. McMillan, "Distilled gpt for source code summarization," *Automated Software Engineering*, vol. 31, no. 1, p. 22, 2024.

[18] E. Shi, Y. Wang, H. Zhang, L. Du, S. Han, D. Zhang, and H. Sun, "Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 39–51.

[19] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[20] B. Yang, H. Tian, J. Ren, H. Zhang, J. Klein, T. F. Bissyandé, C. L. Goues, and S. Jin, "Multi-objective fine-tuning for enhanced program repair with llms," *arXiv preprint arXiv:2404.12636*, 2024.

[21] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[22] "Replication package." [Online]. Available: https://github.com/saimaafrin/QLoRA-Code-Summarization

[23] W. Sun, C. Fang, Y. Chen, Q. Zhang, G. Tao, Y. You, T. Han, Y. Ge, Y. Hu, B. Luo *et al.*, "An extractive-and-abstractive framework for source code summarization," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 3, pp. 1–39, 2024.

[24] E. Shi, Y. Wang, L. Du, J. Chen, S. Han, H. Zhang, D. Zhang, and H. Sun, "On the evaluation of neural code summarization," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 1597–1608.

[25] C. Fang, W. Sun, Y. Chen, X. Chen, Z. Wei, Q. Zhang, Y. You, B. Luo, Y. Liu, and Z. Chen, "Esale: Enhancing code-summary alignment learning for source code summarization," *IEEE Transactions on Software Engineering*, 2024.

[26] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *arXiv preprint arXiv:2005.00653*, 2020.

[27] A. Mastropaolo, L. Pascarella, and G. Bavota, "Using deep learning to generate complete log statements," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2279–2290.

[28] T. Ahmed, K. S. Pai, P. Devanbu, and E. T. Barr, "Improving few-shot prompts with relevant static analysis products," *arXiv preprint arXiv:2304.06815*, 2023.

[29] S. Arakelyan, R. Das, Y. Mao, and X. Ren, "Exploring distributional shifts in large language models for code analysis," in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 16 298–16 314.

[30] F. Chen, F. H. Fard, D. Lo, and T. Bryksin, "On the transferability of pre-trained language models for low-resource programming languages," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 401–412.

[31] J. Gu, P. Salza, and H. C. Gall, "Assemble foundation models for automatic code summarization," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2022, pp. 935–946.

[32] R. Haldar and J. Hockenmaier, "Analyzing the performance of large language models on code summarization," *arXiv preprint arXiv:2404.08018*, 2024.

[33] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[34] R. Anil, A. M. Dai, O. Firat, M. Johnson, D. Lepikhin, A. Passos, S. Shakeri, E. Taropa, P. Bailey, Z. Chen *et al.*, "Palm 2 technical report," *arXiv preprint arXiv:2305.10403*, 2023.

[35] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[36] W. Sun, C. Fang, Y. You, Y. Miao, Y. Liu, Y. Li, G. Deng, S. Huang, Y. Chen, Q. Zhang *et al.*, "Automatic code summarization via chatgpt: How far are we?" *arXiv preprint arXiv:2305.12865*, 2023.

[37] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[38] N. Houlsby, A. Giurgiu, S. Jastrzebski, B. Morrone, Q. De Laroussilhe, A. Gesmundo, M. Attariyan, and S. Gelly, "Parameter-efficient transfer learning for nlp," in *International conference on machine learning*. PMLR, 2019, pp. 2790–2799.

[39] B. Lester, R. Al-Rfou, and N. Constant, "The power of scale for parameter-efficient prompt tuning," *arXiv preprint arXiv:2104.08691*, 2021.

[40] X. L. Li and P. Liang, "Prefix-tuning: Optimizing continuous prompts for generation," *arXiv preprint arXiv:2101.00190*, 2021.

[41] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," *arXiv preprint arXiv:2106.09685*, 2021.

[42] D. Wang, B. Chen, S. Li, W. Luo, S. Peng, W. Dong, and X. Liao, "One adapter for all programming languages? adapter tuning for code search and summarization," in *2023 IEEE/ACM 45th International Conference on Software Engineering*. IEEE, 2023, pp. 5–16.

[43] J. Liu, C. Sha, and X. Peng, "An empirical study of parameter-efficient fine-tuning methods for pre-trained code models," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2023, pp. 397–408.

[44] W. Sun, C. Fang, Y. You, Y. Chen, Y. Liu, C. Wang, J. Zhang, Q. Zhang, H. Qian, W. Zhao *et al.*, "A prompt learning framework for source code summarization," *arXiv preprint arXiv:2312.16066*, 2023.

[45] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," in *Low-Power Computer Vision*. Chapman and Hall/CRC, 2022, pp. 291–326.

[46] X. Zhu, J. Li, Y. Liu, C. Ma, and W. Wang, "A survey on model compression for large language models," *arXiv preprint arXiv:2308.07633*, 2023.

[47] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[48] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," *arXiv preprint arXiv:2204.05999*, 2022.

[49] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[50] L. Tunstall, N. Lambert, N. Rajani, E. Beeching, T. Le Scao, L. von Werra, S. Han, P. Schmid, and A. Rush, "Creating a coding assistant with starcoder," *Hugging Face Blog*, 2023, https://huggingface.co/blog/starchat-alpha.

[51] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.

[52] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[53]

[54] D. Gros, H. Sezhiyan, P. Devanbu, and Z. Yu, "Code to comment" translation" data, metrics, baselining & evaluation," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 746–757.

[55] Y. Majdoub and E. B. Charrada, "Debugging with open-source large language models: An evaluation," *arXiv preprint arXiv:2409.03031*, 2024.

[56] C. Wang, S. Gao, C. Gao, W. Wang, C. Y. Chong, S. Gao, and M. R. Lyu, "A systematic evaluation of large code models in api suggestion: When, which, and how," *arXiv preprint arXiv:2409.13178*, 2024.

[57] Z. Yuan, J. Liu, Q. Zi, M. Liu, X. Peng, and Y. Lou, "Evaluating instruction-tuned large language models on code comprehension and generation," *arXiv preprint arXiv:2308.01240*, 2023.

[58] L. Fan, J. Liu, Z. Liu, D. Lo, X. Xia, and S. Li, "Exploring the capabilities of llms for code change related tasks," *arXiv preprint arXiv:2407.02824*, 2024.

[59] K. I. Roumeliotis, N. D. Tselikas, and D. K. Nasiopoulos, "Llama 2: Early adopters' utilization of meta's new open-source pretrained model," 2023.

[60] "Code llama models at hugging face," https://huggingface.co/codellama.

[61] D. Zan, A. Yu, W. Liu, D. Chen, B. Shen, W. Li, Y. Yao, Y. Gong, X. Chen, B. Guan *et al.*, "Codes: Natural language to code repository via multi-layer sketch," *arXiv preprint arXiv:2403.16443*, 2024.

[62] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang, "Universal fuzzing via large language models," *arXiv preprint arXiv:2308.04748*, 2023.

[63] T. B. Brown, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[64] T. Dettmers, M. Lewis, S. Shleifer, and L. Zettlemoyer, "8-bit optimizers via block-wise quantization," *arXiv preprint arXiv:2110.02861*, 2021.

[65] C. Wu and M. Yan, "Learning deep semantic model for code search using codesearchnet corpus," *arXiv preprint arXiv:2201.11313*, 2022.

[66] A. Mastropaolo, E. Aghajani, L. Pascarella, and G. Bavota, "An empirical study on code comment completion," in *2021 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2021, pp. 159–170.

[67] J. Chen, X. Hu, Z. Li, C. Gao, X. Xia, and D. Lo, "Code search is all you need? improving code suggestions with code search," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[68] D. Roy, S. Fakhoury, and V. Arnaoudova, "Reassessing automatic evaluation metrics for code summarization tasks," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1105–1116.

[69] C. Zhang, J. Wang, Q. Zhou, T. Xu, K. Tang, H. Gui, and F. Liu, "A survey of automatic source code summarization," *Symmetry*, vol. 14, no. 3, p. 471, 2022.

[70] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th international conference on program comprehension*, 2020, pp. 184–195.

[71] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[72] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments," in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005, pp. 65–72.

[73] C.-Y. Lin, "Rouge: A package for automatic evaluation of summaries," in *Text summarization branches out*, 2004, pp. 74–81.

[74] M. Popović, "chrf: character n-gram f-score for automatic mt evaluation," in *Proceedings of the tenth workshop on statistical machine translation*, 2015, pp. 392–395.

[75] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, "Bertscore: Evaluating text generation with bert," *arXiv preprint arXiv:1904.09675*, 2019.

[76] J. Devlin, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[77] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[78] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[79] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian journal of statistics*, pp. 65–70, 1979.

[80] K. Krippendorff, "Reliability in content analysis: Some common misconceptions and recommendations," *Human communication research*, vol. 30, no. 3, pp. 411–433, 2004.

[81] M. Abdin, J. Aneja, H. Awadalla, A. Awadallah, A. A. Awan, N. Bach, A. Bahree, A. Bakhtiari, J. Bao, H. Behl *et al.*, "Phi-3 technical report: A highly capable language model locally on your phone," *arXiv preprint arXiv:2404.14219*, 2024.

[82] X. Deng, S. Zhong, H. Dong, J. Hu, S. M. Beillahi, X. Si, and F. Long, "Assessing code generation with intermediate languages," *arXiv preprint arXiv:2407.05411*, 2024.

[83] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[84] M. D. L. C. Peixoto, D. d. M. Baia, N. Nascimento, P. Alencar, B. Fonseca, and M. Ribeiro, "On the effectiveness of llms for manual test verifications," *arXiv preprint arXiv:2409.12405*, 2024.

[85] S. Haque, Z. Eberhart, A. Bansal, and C. McMillan, "Semantic similarity metrics for evaluating source code summarization," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 36–47.

[86] Y. Virk, P. Devanbu, and T. Ahmed, "Enhancing trust in llm-generated code summaries with calibrated confidence scores," *arXiv preprint arXiv:2404.19318*, 2024.

[87] J. Zhu, Y. Miao, T. Xu, J. Zhu, and X. Sun, "On the effectiveness of large language models in statement-level code summarization," in *2024 IEEE 24th International Conference on Software Quality, Reliability and Security*. IEEE, 2024, pp. 216–227.

[88] J. Yang, B. Hui, M. Yang, J. Yang, J. Lin, and C. Zhou, "Synthesizing text-to-sql data from weak and strong llms," *arXiv preprint arXiv:2408.03256*, 2024.

[89] J. Lio, G. Li, Y. Li, and Z. Jin, "Structured chain-of-thought prompting for code generation," *ACM Transactions on Software Engineering and Methodology*, 2023.