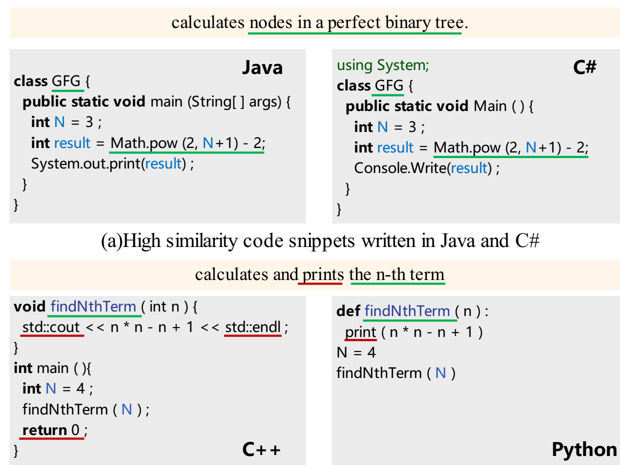
Mixture-of-Experts Low-Rank Adaptation for Multilingual Code Summarization

Tianchen Yu[1,2*], Li Yuan[1,2*], Hailing Huang[1,2], Jiexin Wang[1,2], Yi Cai[2,1†]
[1] School of Software Engineering, South China University of Technology, Guangzhou, China
[2] Key Laboratory of Big Data and Intelligent Robot(SCUT), MOE of China
{seyutianchen,seyuanli}@mail.scut.edu.cn, hhl1132714358@gmail.com,{jiexinwang, ycai}@scut.edu.cn

***Abstract*—As Code Language Models (CLMs) are increasingly used to automate multilingual code intelligence tasks, Full-Parameter Fine-Tuning (FPFT) of CLMs has become a widely adopted approach, which is both time-consuming and resource-intensive. Parameter-Efficient Fine-Tuning (PEFT) provides a more efficient alternative to FPFT. However, it struggles to capture common features shared across languages, leading to performance degradation. Recent studies have explored mixed-language training with PEFT to avoid the loss of common features. However, these methods can result in gradient conflicts due to the diverse language-specific features, causing suboptimal performance, particularly for low-resource languages. In this paper, we propose Mixture-of-Experts Multilingual Low-Rank Adaptation (MMLoRA) for multilingual code summarization. MMLoRA addresses gradient conflicts while preserving common features shared across languages by combining a universal expert with a set of specialized linguistic experts. Additionally, we introduce an expert loss function that maintains the diversity of specialized linguistic experts while balancing the learning progress. Experimental results indicate that MMLoRA achieves state-of-the-art performance in multilingual code summarization while maintaining efficient fine-tuning. The performance improvement is particularly significant in low-resource languages such as Ruby.**

***Index Terms*—Code Summarization, Low-Rank Adaptation, Mixture-of-Experts**



(a)High similarity code snippets written in Java and C#

(b)Low similarity Code snippets written in C++ and Python

Fig. 1. Two pairs of functionally identical code snippets are provided in different programming languages. In (a), the code written in Java and C# demonstrates many common features. In (b), the code snippets written in C++ and Python each showcase their language-specific features.

I. Introduction

Software engineers often face the challenge of handling multiple programming languages during the design and development phases [1], [2]. Multilingual code summarization is one of the most representative multilingual tasks [3]–[5], aiming to learn common features shared across multiple languages to enhance the code summarization capabilities, particularly for low-resource languages [6]. Moreover, multilingual code summarization facilitates the evaluation of a model's capability for multilingual code comprehension through human evaluation [7]–[11], as it requires the model to capture essential common features of the entire code, including keywords, syntactic structures, and variable names [12], [13]. However, traditional approaches [14]–[16] often struggle to capture such common language features due to the inherent performance limitations of their base models.

Recently, Code Language Models (CLMs) represented by CodeGen [17], StarCoder [18], and DeepSeekCoder [19] have been introduced. Leveraging their extensive world knowledge and large-scale parameterization, CLMs have demonstrated remarkable performance across various software engineering tasks. Although CLMs are designed to generalize across diverse codebases, they still require fine-tuning to effectively adapt to multilingual code summarization tasks [20]. A straightforward approach to integrating CLMs into multilingual code summarization is Full-Parameter Fine-Tuning (FPFT). However, for multilingual tasks, FPFT demands substantial training time and computational resources to optimize vast number of model parameters [20], [21], making it challenging to implement in real-world scenarios.

Parameter-Efficient Fine-Tuning (PEFT) [21] has emerged as an efficient approach to overcome the limitations of FPFT, reducing computational costs by training only a small subset of model parameters while keeping the majority frozen. Low-Rank Adaptation (LoRA) [22] is one of the most popular PEFT techniques and has been successfully applied to various downstream tasks, including software engineering tasks [23] such as code repair [24] and code generation [25]. Although LoRA significantly reduces resource overhead compared to FPFT, maintaining separate LoRA models for a large number of programming languages can still become resource-intensive. Moreover, individual LoRA models fail to leverage

* Equal contribution.
† Corresponding Author

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00186

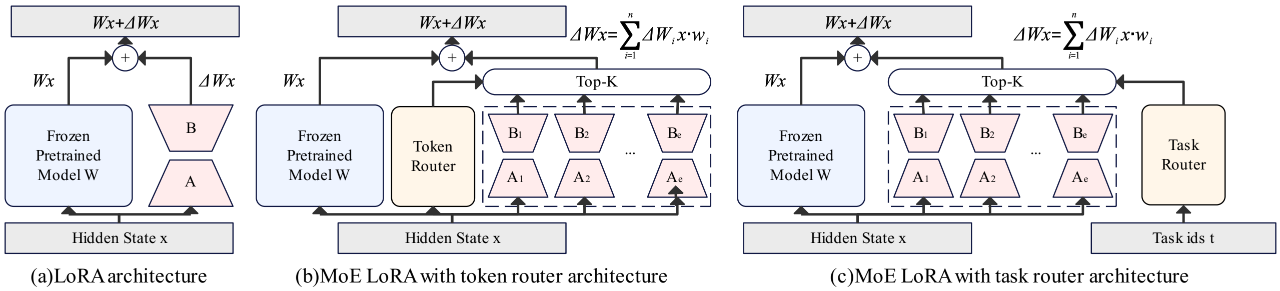


Fig. 2. Different LoRA architectures. (a) is the vanilla LoRA architecture, (b) is the LoRA with MoE architecture that uses a token router, and (c) is the LoRA with MoE architecture which uses a task router.

the common features that shared across languages, which are believed to enhance a model's understanding of the characteristics inherent to different languages [1], [20]. Consequently, this may lead to suboptimal performance, particularly for low-resource languages. For example, Fig. 1(a) shows two functionally equivalent code snippets written in **Java** and **C#**. Despite being in different languages, they exhibit a high degree of similarity in terms of variable names, syntax, and keywords. Such common features shared across **Java** and **C#** enable a programmer proficient in **Java** to also understand **C#**. Furthermore, models need to learn these common features in order to generate accurate code summarizations.

To learn the common features shared across languages, a universal LoRA can be used for mixed multilingual training [1], [20], [26], [27]. However, some studies [28], [29] indicate that mixed multilingual training can lead to gradient conflicts [30], [31], as the model produces opposing gradients when processing different natural languages as input. Such cases are also common in programming languages. Fig. 1(b) presents two functionally equivalent code snippets in **Python** and **C++**. Although both snippets achieve the same functionality, they differ in indentation style and output formatting. Besides, when a dataset is dominated by **C++** and contains relatively few **Python** examples, gradient conflicts can lead the model to become biased toward the high-resource language. This bias causes negative knowledge transfer [6], resulting in suboptimal performance on the low-resource language.

To address these challenges, we propose Mixture-of-Experts Multilingual Low-Rank Adaptation (MMLoRA) for multilingual code summarization. MMLoRA uses LoRA as the foundation and enhances the model's ability to deeply understand multilingual data. Specifically, MMLoRA distinguishes itself by extending the Mix-of-Experts (MoE) [32]–[38] architecture through three key aspects, comprising a universal expert, a set of specialized linguistic experts, and a routing strategy with an expert loss function. First, a universal expert is introduced to learn common features shared across multiple languages, facilitating cross-linguistic knowledge transfer. Meanwhile, we establish a set of specialized linguistic experts that focus on capturing language-specific features to mitigate gradient conflicts caused by distinct features across languages. Two types of experts are complementary in MMLoRA. Finally, to better integrate the universal expert and specialized linguistic experts, we propose a routing strategy with an expert loss function. This loss function maintains diversity among the specialized linguistic experts while ensuring stable and efficient learning.

We evaluate MMLoRA on multilingual code summarization across two widely adopted multilingual code summarization datasets [39], [40]. The experimental results indicate that MMLoRA effectively learned both common features and language-specific features, demonstrating strong improvements in low-resource languages such as Ruby.

The contributions of this paper are as follows:

- We extend the MoE structure by utilizing a universal expert and a set of specialized linguistic experts. The universal expert retains common features to enhance understanding of low-resource languages, while the specialized linguistic experts capture language-specific features to mitigate gradient conflicts.
- We propose an expert loss function that includes a diversity loss and a balanced loss to ensure differentiation among specialized linguistic experts while maintaining a balanced learning pace.
- Based on the pre-trained StarCoderBase and DeepSeek-Coder models, MMLoRA achieves state-of-the-art results on the code summarization task while requiring training on a small percentage of the total parameters. MMLoRA particularly achieves significant improvements in low-resource languages such as Ruby, with an average performance increase of 8% compared to vanilla LoRA.

II. Background

A. Parameter-Efficient Fine-Tuning

As the parameter scale of CLMs continues to expand, the cost of fine-tuning all parameters has become increasingly prohibitive. To address this challenge, Parameter-Efficient Fine-Tuning methods have been introduced, with popular approaches including Prefix Tuning [21], Adapter Tuning [41], Prompt Tuning [42], and LoRA [22]. Prefix-tuning [21] optimizes a small set of task-specific vectors to enable effective task specialization with minimal parameter adjustments. Adapter Tuning [41] enables fine-tuning by inserting small trainable neural networks between each layer of the model. In Prompt Tuning [42], trainable prefix embeddings are added before the input layer to influence the model's outputs.

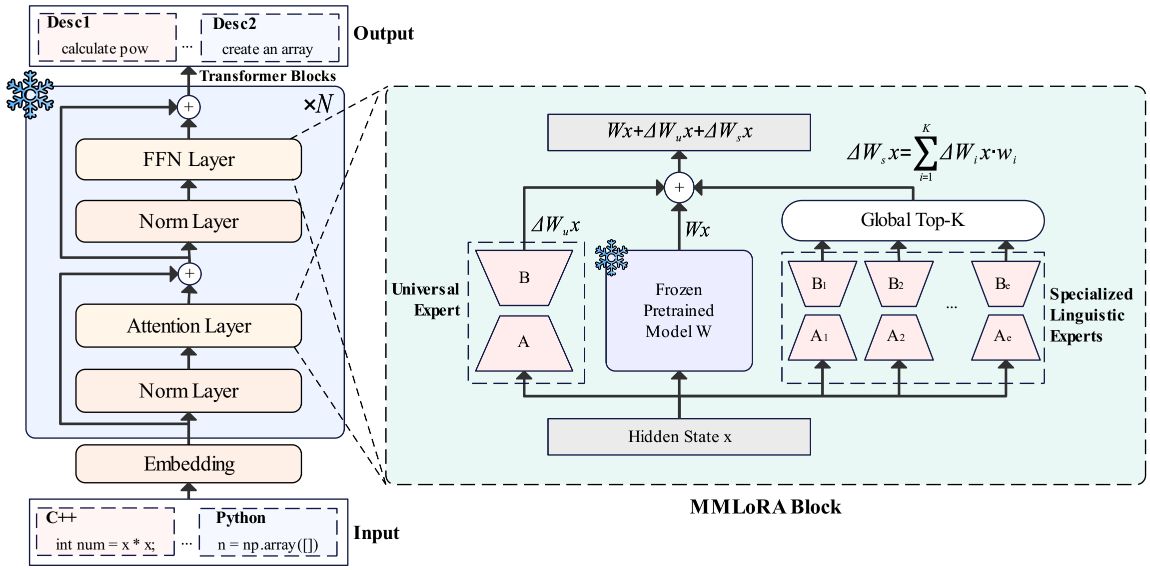


Fig. 3. Overall architecture of our proposed MMLoRA. The linear layers in the FFN Layers and Attention Layers are replaced by the MMLoRA Block, which consists of a universal expert and multiple specialized linguistic experts. Each expert functions as a LoRA module.

LoRA [22] adapts to new tasks by integrating trainable low-rank matrices within the model's weight matrices. Fig. 2(a) provides an architecture overview of LoRA, where A and B are two newly added trainable matrices. LoRA has emerged as one of the most widely adopted PEFT methods due to its low complexity, high performance and strong scalability. Numerous studies have refined LoRA for further efficiency. For example, DoRA [43] decomposes pre-trained weights into two components and applies LoRA for directional updates during fine-tuning. Tied-LoRA [44] introduces weight tying, further reducing the number of trainable parameters. AdaLoRA [45] employs Singular Value Decomposition to decompose matrices, allowing for more streamlined updates. Recent studies [23], [24], [46], [47] have also applied LoRA to software engineering tasks.

B. *Mixture-of-Experts*

Mixture-of-Experts [48] is an innovative supervised learning framework featuring multiple specialized experts, each designed to process specific subsets of training data. MoE modifies the linear layers within transformer blocks by introducing experts with sparse activations, allowing for increased model capacity without a rise in computational cost. To facilitate differentiation among experts, MoE architectures often utilize sampling strategies and routing mechanisms.

Recent research on fine-tuning pretrained models has successfully integrated MoE with LoRA fine-tuning, achieving strong results across multiple fields. The differentiation among LoRA with MoE architectures primarily depends on the routing mechanisms, which fall into two main approaches. Fig. 2(b) shows the LoRA with MoE architecture using a token router, which assigns appropriate experts to each token within the input sequence. Fig. 2(c) shows the LoRA with MoE architecture using a task router, which uses the task type to determine which experts to apply to a given input. MOELoRA [32] and MoA [33] both employ a task router to facilitate efficient multi-task learning. LoRAMoE [34] and MoCLE [35] leverages a token router to prevent the forgetting of world knowledge. MOLA [49] introduces a method that allocates a greater number of experts to deeper model layers to investigate the effect of expert numbers on model performance.

III. Our Approach

In this section, we first introduce the overall architecture of MMLoRA as shown in Fig. 3. In our approach, we replace the linear layers in the Feed-Forward Network (FFN) and the Attention Layer of the Transformer blocks with our proposed MMLoRA Block. Each MMLoRA Block comprises a universal expert and a set of specialized linguistic experts. To balance high performance and fine-tuning efficiency, the universal expert remains active throughout the training process, while specialized linguistic experts are selectively activated according to the global Top-K strategy.

A. *Low-rank Adaptation*

Low-rank Adaptation [22] is an efficient approach for fine-tuning pretrained models, which serves as the foundation for both the universal and specialized linguistic experts in our MMLoRA architecture. Thus, we first outline the core mechanism of LoRA. Given a frozen pre-trained weight matrix $W \in \mathbb{R}^{d\times k}$, LoRA introduces two trainable matrices, $A \in \mathbb{R}^{d\times r}$ and $B \in \mathbb{R}^{r\times k}$, where $r < \min(d, k)$. The same input is multiplied by both W and the $\Delta W = BA$ weight adjustments of each expert, which is defined as follows:

$$h = Wx + \Delta Wx = Wx + BAx \tag{1}$$

where Wx is the output of the original linear layer, ΔWx is the output of LoRA, and h is the final output result.

B. MMLoRA Block Architecture

MMLoRA consists of multiple pairs of low-rank matrices, with each pair referred to as an expert. Specifically, each MMLoRA block comprises a universal expert and a set of specialized linguistic experts. The universal expert remains activated throughout the training process, while the specialized linguistic experts are dynamically selected for activation based on the input language. MMLoRA employs a simple Global Language Router to facilitate this dynamic selection, which executes a Global Top-K selection to activate the most relevant specialized experts across all MMLoRA blocks.

Since each input is assigned a fixed language ID, designing a dedicated router for each language in every MMLoRA block would significantly increase both the number of trainable parameters and computational overhead. To address this issue, we implement a single Global Language Router that allocates weights to the specialized linguistic experts, thereby minimizing redundant trainable parameters and enhancing computational efficiency. The Global Language Router consists only of an embedding layer and a Multilayer Perceptron (MLP), where each input includes a language ID t. The Global Language Router then produces the following output distribution:

$$\begin{aligned} R_l &= \max(0, \text{Dropout}(\text{Embedding}(t)W_1))W_2 \\ S_l &= \text{Softmax}(R_l) \end{aligned} \tag{2}$$

where W_1 and W_2 represent two trainable weight matrices in the MLP, R_l is the output routing logits, and S_l is the output routing score. Global Language Router is a single module for the entire model. Therefore, the Global Top-K at different positions of the MMLoRA Block will share the same routing logits R_l and routing score S_l.

The universal expert is always selected and trained, while specialized linguistic experts are chosen based on the routing score S_l using Global Top-K. This selection process can be represented as follows:

$$w_i = \frac{\text{Topk}(S_l, K)_i}{\sum_{i=1}^{K} \text{Topk}(S_l, K)_i} \tag{3}$$

where K is the number of selected experts, and w_i represents the expert weight calculated for the i-th selected specialized linguistic expert. The sum of the expert weights for all selected specialized linguistic experts is equal to 1. The expert weights of the remaining unselected specialized linguistic experts are set to 0 and will not be called during the forward pass. For a linear layer, the forward process is defined as $h = Wx$. Therefore, the final forward propagation of the MMLoRA is represented as follows:

$$h = Wx + \Delta W_u x + \sum_{i=1}^{K} \Delta W_i x \cdot w_i \tag{4}$$

where W_u is the trainable weights for the universal expert, W_i and w_i represent the trainable weights and expert weight of the i-th selected specialized linguistic expert, x is the input to the MMLoRA Block, and h is the output of the MMLoRA Block, which is passed to the subsequent model layer.

The model outputs a sequence of tokens, with the loss function defined as cross-entropy loss, which measures the discrepancy between the predicted probability distribution and the actual probability distribution of the tokens. The cross-entropy loss is given by:

$$\mathcal{L}_l = -\sum_{t=1}^{T} \log P(x_t \mid x_1, x_2, \ldots, x_{t-1}) \tag{5}$$

where T is the length of the sequence, x_t is the token at position t, and $P(x_t \mid x_1, \ldots, x_{t-1})$ is the predicted probability for the token x_t conditioned on the previous tokens.

C. Expert Loss Function

To resolve the imbalance in expert allocation and improve the model's ability to learn both common and language-specific features, we propose a novel expert loss function based on the Global Language Router. Notably, previous MoE loss functions [36], [50] often prioritize distribution balance across experts without ensuring adequate differentiation among them. Our loss function consists of two parts: a diversity loss that measures the degree of diversity in the router weights assigned to experts for different languages, and a balanced loss that calculates the standard deviation of the cumulative router weights distributed across all experts.

The Diversity Loss $\mathcal{L}_d$ is designed to promote a diverse allocation of specialized linguistic experts across different languages. To calculate $\mathcal{L}_d$, a list $\hat{t}$ containing all language IDs is input to the Global Language Router at each training step. The Global Language Router then outputs router scores S_t, which is used in Equation (3) to compute the weights assigned to each expert for each language. The weight of the m-th specialized linguistic expert for the n-th language is w_n^m, and the weights of all experts for the n-th language can be represented as a vector $V_n \in [w_n^1, w_n^2, \ldots, w_n^m]$. The $\mathcal{L}_d$ can be represented as follows:

$$\mathcal{L}_d = \frac{2 \cdot \sum_{i=1}^{N} \sum_{j=i+1}^{N} \sum_{m=1}^{M} (V_i \cdot V_j)_m}{N(N-1)} \tag{6}$$

where N is the number of languages, and M is the number of specialized linguistic experts. A smaller value of $\mathcal{L}_d$ indicates greater differentiation among the experts.

Besides, to ensure a balanced weight distribution among experts. We propose a balance loss $\mathcal{L}_b$. Specifically, the sum of the weights of all languages for the m-th specialized linguistic expert $w_m = \sum_{i=1}^{N} w_i^m$. Thus, the weights of all experts can be represented as a vector $\hat{V} \in [w_1, w_2, \ldots, w_m]$. The $\mathcal{L}_b$ can be represented as follows:

$$\mathcal{L}_b = \sqrt{\frac{1}{M} \sum_{i=1}^{M} (\hat{V}_i - \mu_{\hat{V}})^2} \tag{7}$$

where $\mu_{\hat{V}}$ is the mean of the vector $\hat{V}$ and $\hat{V}_i$ is the i-th element of the vector $\hat{V}$. A smaller value of $\mathcal{L}_b$ indicates a more balanced learning progress across the experts.

TABLE I
PRETRAINING DATA PROPORTION AND CODE SUMMARIZATION DATASETS

Prog Language	Pretraining Data Proportion†		CodeSearchNet			XLCoST		
	DeepSeekCoder	StarCoderBase	Train	Validation	Test	Train	Validation	Test
Java	18.63%	11.33%	164,923 (18.16%)	5,183	10,955	9,623 (19.18%)	911	494
C++	11.39%	6.38%	-	-	-	9,797 (19.53%)	909	492
Python	15.12%	7.87%	251,820 (27.73%)	13,914	14,918	9,263 (18.46%)	887	472
C#	7.34%	5.82%	-	-	-	9,345 (18.63%)	899	491
PHP	7.38%	7.93%	241,241 (26.56%)	12,982	14,014	3,087 (6.15%)	308	158
Go	0.32%	3.10%	167,288 (18.42%)	7,325	8,122	-	-	-
Javascript	6.75%	8.44%	58,025 (6.39%)	3,885	3,291	8,590 (17.12%)	886	475
C	3.59%	7.03%	-	-	-	463 (0.92%)	51	60
Ruby	1.88%	0.89%	24,927 (2.74%)	1,400	1,261	-	-	-

† denote the result reported from DeepSeekCoder [19] and StarCoder [18].

Finally, we obtain the Expert Loss $\mathcal{L}_e$ by taking a weighted sum of the Diversity Loss $\mathcal{L}_d$ and the Balanced Loss $\mathcal{L}_b$. The loss function can be represented as follows:

$$\begin{aligned} \mathcal{L}_e &= c_d \cdot \mathcal{L}_d + c_b \cdot \mathcal{L}_b \\ \mathcal{L} &= \mathcal{L}_e + \mathcal{L}_l \end{aligned} \tag{8}$$

where c_d is the weight of the Diversity Loss, and c_b is the weight of the Balanced Loss. $\mathcal{L}$ is the final loss used for training.

IV. EXPERIMENTAL SETTINGS

This paper employs multilingual code summarization to evaluate MMLoRA's performance on the multilingual task. We will next describe the datasets, evaluation metrics, and implementation details.

A. Dataset Details

Code summarization focuses on generating natural language descriptions that explain the functionality of given code snippets. We utilize two multilingual code summarization datasets, specifically CodeSearchNet [39] from CodeXGLUE [51] and XLCoST [40]. Table I shows the distribution of pretraining data across the two base models, together with detailed statistics of two datasets. Further details regarding the two datasets are provided below:

1) **CodeSearchNet** [39]: CodeSearchNet is a widely used dataset for multilingual analysis [1], [20], [23], [52], which provides pairs of code snippets and their corresponding natural language summaries across six programming languages: Python, Java, JavaScript, Ruby, Go, and PHP. Notably, there are substantial differences in data volume across these languages. First, the proportion of pretraining code data for each language in the base models varies significantly. Second, the amount of fine-tuning data provided by CodeSearchNet is also highly imbalanced, which further exacerbating the challenge of learning the features of these low-resource languages. A typical example is Ruby, which accounts for less than 2% of the pretraining data in the base model and only 2.74% of the fine-tuning data, making it the lowest-resource language among the six languages.
2) **XLCoST** [40]: XLCoST is a multilingual code summarization dataset that covers seven programming languages, namely Java, C++, Python, C#, PHP, JavaScript, and C. Similar to CodeSearchNet, XLCoST also exhibits significant data imbalance across different languages. For example, C is a typical low-resource language, as it contains only less than 1% of the fine-tuning data.

B. Evaluation Metrics

We use two commonly used metrics Bleu [53] and Meteor [54] to evaluate the model's performance on code summarization. Detailed information about the evaluation metrics is as follows:

1) **Bleu** [53]: Bleu quantifies n-gram overlap between generated and reference text, which is commonly used to evaluate the quality of generated text in machine translation and summarization tasks. Higher Bleu scores indicates that the generated results are of higher quality. In our experiments, we directly apply the Bleu calculation function provided by CodeXGLUE [51], which is also employed by several other approaches [1], [20].
2) **Meteor** [54]: Meteor is another widely used metric for evaluating the quality of generated summaries. Meteor prioritizes semantic similarity and linguistic flexibility, achieving stronger alignment with human evaluations [55] by accounting for morphological variations and synonym matching. A higher Meteor score indicates higher quality output.

C. Implementation Details

We chose DeepSeekCoder-1.3B [19] and StarCoderBase-1.1B [18] as the base models for evaluating different fine-tuning methods. For LoRA fine-tuning, we set the rank to 32 and LoRA Alpha to 64. For MoLA [49], we followed the original paper's settings The number of experts for each group was configured incrementally from the bottom up, set to 2, 4, 6 and 8, with the top-K parameter K set to 2 and rank set to 8 for all groups. In our MMLoRA configuration, the number of experts was set to 4, with top-K K set to 2. The universal expert rank was set to 8, and LoRA Alpha to 16. The specialized linguistic experts had a rank of 4 and LoRA Alpha of 8. We applied a Dropout rate of 0.1 across all LoRA modules. In Global Language Router, the embedding layer's input dimension was based on the number of tasks, with an output dimension set to 128. The MLP layer had an intermediate dimension of 512, a Dropout rate of 0.1, and an output dimension equal to the number of experts.

TABLE II
RESULTS OF CODE SUMMARIZATION COMPARISON WITH OTHER METHODS ON CODESEARCHNET

Data Availability			High-resource ⟶									⟶Low-resource			Overall	
Models	Methods	Param(%)	Java		Python		PHP		Go		Javascript		Ruby			
			Bleu	Meteor	Bleu	Meteor	Bleu	Meteor	Bleu	Meteor	Bleu	Meteor	Bleu	Meteor	Bleu	Meteor
GPT-4	one-shot	-	9.22	29.25	8.21	32.16	8.96	31.12	8.49	29.18	7.00	19.96	6.29	22.06	8.03	27.29
	few-shot	-	10.93	31.20	12.08	33.48	12.20	33.81	12.21	35.85	9.35	20.71	10.01	24.41	11.13	29.91
Deepseekv3	one-shot	-	8.39	30.07	8.02	32.04	7.79	29.02	8.46	27.45	6.52	19.84	6.14	23.38	7.55	26.97
	few-shot	-	13.83	34.90	13.61	33.50	13.34	36.39	12.72	32.11	10.84	20.10	10.29	24.92	12.04	30.32
Deep SeekCoder	FPFT-Mix	100%	19.48	33.25	19.85	32.22	23.93	35.18	19.83	33.90	13.05	20.10	14.78	22.85	18.49	29.58
	LoRA-Each	2.17%×6	21.20	35.45	20.16	33.36	25.61	35.75	**21.77**	37.78	13.67	20.24	15.06	24.28	19.58	31.14
	LoRA-Mix	2.17%	**21.70**	35.29	20.00	32.62	25.56	35.82	21.06	37.35	13.30	20.53	15.15	23.90	19.46	30.92
	MoLA	3.03%	20.78	34.86	20.35	32.53	26.61	35.71	21.57	36.78	**14.21**	20.18	15.04	23.00	19.60	31.51
	MMLoRA	1.65%	21.38	**35.65**	**20.95**	**34.13**	**26.83**	**38.08**	21.75	**38.25**	13.80	**20.96**	**16.55**	**25.65**	**20.19**	**32.12**
	p-value†		0.159	0.007	0.035	<0.001	<0.001	<0.001	<0.001	<0.001	0.035	<0.001	<0.001	<0.001	<0.001	<0.001
Star CoderBase	FPFT-Mix	100%	20.05	33.57	18.71	29.75	22.34	33.31	19.02	34.31	13.66	19.17	14.34	22.93	18.02	28.81
	LoRA-Each	1.91%×6	22.93	35.49	19.46	31.58	25.86	38.03	21.89	37.17	12.54	19.36	14.95	23.97	19.61	30.93
	LoRA-Mix	1.91%	21.81	33.71	19.65	32.70	**26.34**	**38.35**	20.94	36.51	14.33	21.03	16.01	24.48	19.85	31.13
	MoLA	2.68%	22.30	34.32	20.22	**33.61**	25.21	37.67	**22.76**	**38.25**	14.13	21.08	15.66	25.72	20.05	31.78
	MMLoRA	1.56%	**23.47**	**35.59**	**20.28**	32.74	25.19	37.30	21.78	37.14	**14.46**	**21.98**	**16.35**	**25.88**	**20.26**	**31.91**
	p-value†		<0.001	<0.001	<0.001	0.105	0.098	0.102	0.024	0.117	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001

The p-value † corresponds to the comparison between MMLoRA and MoLA.

Regarding training details, the PEFT method was applied to all MLP layers and attention layers. For all tasks, the learning rate across all fine-tuning methods was set to 0.0001, with a batch size of 24 and a warm-up of 100 steps. The training epoch is set to 1 in normal scenarios and 2 in low-resource scenarios. All experiments were conducted using 4 RTX 4090 GPUs, and each card has 24GB of memory.

V. EXPERIMENTAL RESULTS

A. RQ1: How effective is MMLoRA on multilingual code summarization?

To assess the effectiveness of MMLoRA, we compared it against LoRA [22] and MoLA [49] using two CLMs: DeepSeekCoder-1.3B [19] and StarCoderBase-1.1B [18]. We focus on CLMs with approximately 1B parameters, as these models are frequently deployed in resource-constrained environments and are vulnerable to imbalances in resource allocation [56]. Besides, we evaluated GPT-4 and Deepseek-v3 on 50 samples per language from CodeSearchNet and XLCoST using both one-shot and few-shot prompts. For the few-shot setting, each prompt included four randomly selected examples following a previous study [10]. Specifically, we compared MMLoRA with four fine-tuning approaches:

- **FPFT-Mix**: This method combines data from all languages to fine-tune the CLMs using the FPFT technique. We did not evaluate selecting FPFT individually for each language, as this would result in high computational costs, making it unfeasible for real-world applications.
- **LoRA-Each**: This method applies separate LoRA weights for each language individually.
- **LoRA-Mix**: This method combines data from all languages and employs a single set of LoRA weights for the training process.
- **MoLA** [49]: A LoRA with MoE architecture that employs a token router, allocating additional experts to higher network layers.

The experimental results on CodeSearchNet and XLCoST are presented in Tables II and III respectively, with the best results highlighted in bold and the second-best results underlined. Experimental results indicate that MMLoRA outperformed FPFT and PEFT methods on both models in code summarization. Specifically, on the CodeSearchNet dataset, MMLoRA improved overall Bleu by **10.8%** and Meteor by **9.6%** compared to FPFT-Mix on average. On the XLCoST dataset, MMLoRA demonstrated even larger gains over FPFT-Mix, achieving average overall Bleu improvements of **19.7%** and Meteor improvements of **10.5%** across both models. FPFT-Mix suffers from significant negative knowledge transfer and forgetting of world knowledge, resulting in inferior performance compared to MMLoRA and other PEFT methods on multilingual code summarization. MMLoRA addresses these challenges by effectively fine-tuning language-specific subsets of model parameters, thereby achieving superior performance.

When compared with the current state-of-the-art PEFT method MoLA, MMLoRA still demonstrates superior performance. Statistical analysis reveals the p-values are **less than 0.001** for most programming languages, indicating that the performance improvements are statistically significant. This improvement can be attributed to MMLoRA's incorporation of an additional universal expert, which enhances the model's ability to capture common language features while mitigating gradient conflicts and promoting effective knowledge transfer from high-resource languages to low-resource languages.

Additionally, according to the proportions of pretraining and fine-tuning data shown in Table I, programming languages can be categorized into well-trained high-resource languages (e.g. Java, C++, Python) and low-resource languages with limited data availability (e.g. JavaScript, Ruby, C). For the representative low-resource language Ruby in CodeSearchNet, MMLoRA outperforms MoLA with average gains of **6.5%** in Bleu and **6.3%** in Meteor across two models. Compared with LoRA-Mix, MMLoRA achieves improvements of **5.7%**

TABLE III
RESULTS OF CODE SUMMARIZATION COMPARISON WITH OTHER METHODS ON XLCOST

Data Availability			High-resource → Low-resource														Overall	
Models	Methods	Param(%)	Java		C++		Python		C#		PHP		Javascript		C			
			Bleu	Meteor	Bleu	Meteor	Bleu	Meteor	Bleu	Meteor	Bleu	Meteor	Bleu	Meteor	Bleu	Meteor	Bleu	Meteor
GPT-4	one-shot	-	7.31	29.79	7.79	34.35	7.93	35.57	8.15	33.37	9.41	33.66	6.94	30.54	12.73	41.87	8.61	34.17
	few-shot	-	13.43	34.46	13.55	36.03	14.35	35.09	11.78	31.61	11.71	34.49	11.50	31.58	15.31	43.35	13.09	35.23
Deepseekv3	one-shot	-	10.19	34.14	9.79	32.26	9.96	35.60	10.91	33.69	11.00	33.32	8.39	29.99	14.51	42.17	10.68	34.45
	few-shot	-	11.27	35.31	13.83	35.60	14.23	36.08	12.37	33.35	12.05	34.99	11.66	32.13	19.98	43.60	13.63	35.87
Deep SeekCoder	FPFT-Mix	100%	19.16	32.03	19.11	33.59	18.98	31.89	19.92	31.48	20.82	29.60	18.77	29.66	23.64	33.17	19.62	31.52
	LoRA-Each	2.17%×6	22.66	**36.24**	22.61	35.31	22.25	35.30	**22.92**	**34.66**	**25.59**	34.92	20.72	31.90	21.37	32.77	22.62	34.62
	LoRA-Mix	2.17%	19.54	32.12	19.91	34.04	19.77	32.75	20.86	31.98	21.54	30.36	18.73	30.24	23.95	33.97	20.20	32.07
	MoLA	3.03%	22.32	35.32	22.44	34.41	**23.34**	35.42	21.89	34.32	23.55	34.70	18.96	29.89	27.12	39.78	22.28	34.27
	MMLoRA	1.65%	**23.18**	35.83	**22.66**	**36.20**	22.82	**35.72**	22.27	34.64	25.36	**35.17**	**21.07**	**32.05**	**31.87**	**43.97**	**23.25**	**35.37**
	p-value†		<0.001	0.003	0.028	<0.001	0.011	0.058	0.025	<0.001	0.035	0.130	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
Star CoderBase	FPFT-Mix	100%	19.16	31.23	20.25	32.71	20.02	32.66	19.03	31.30	18.88	31.82	18.04	29.94	22.51	35.95	19.40	31.82
	LoRA-Each	1.91%×6	22.18	34.86	22.39	34.69	20.49	32.26	**21.78**	**33.11**	26.87	36.23	**19.76**	30.40	20.53	25.42	22.00	33.10
	LoRA-Mix	1.91%	19.59	31.82	20.87	33.24	20.85	33.40	19.91	31.77	19.18	32.02	18.28	30.29	22.88	36.25	19.95	32.29
	MoLA	2.68%	22.04	35.04	22.18	34.17	23.00	**35.18**	21.42	32.93	23.51	35.65	18.87	29.85	26.82	39.43	22.01	34.03
	MMLoRA	1.56%	**22.59**	**35.14**	**22.52**	**35.22**	**23.42**	35.03	21.65	33.10	**27.20**	**36.44**	19.44	**31.23**	**30.49**	**41.21**	**23.06**	**34.59**
	p-value†		<0.001	0.104	0.146	0.030	0.092	0.277	0.252	0.312	<0.001	<0.001	0.247	<0.001	<0.001	<0.001	<0.001	<0.001

The p-value † corresponds to the comparison between MMLoRA and MoLA.

in Bleu and **4.7%** in Meteor. When evaluated with Bleu on Ruby, MMLoRA achieves gains of **+1.51** on DeepSeekCoder and **+0.69** on StarCoderBase. In contrast, prior approaches reported much smaller improvements, such as **+0.46** (Multilingual Adapter vs. UniXcoder [20]), **+0.26** (Multilingual Adapter vs. CodeT5 [20]), and **+0.14** (ESALE vs. UniXcoder [7]). These results demonstrate that MMLoRA provides significant improvements on low-resource language Ruby. For the representative low-resource language C in XLCoST, MMLoRA achieved particularly substantial improvements. Specifically, it increased Bleu and Meteor by **15.6%** and **7.5%** compared to MoLA on average, and achieved an average improvement of **more than 20%** compared to LoRA-Mix. For low-resource languages such as JavaScript, which possess limited fine-tuning data but a reasonable amount of pre-training data, the improvements were less pronounced. These findings suggest that MMLoRA more effectively captures both common and language-specific features, thereby facilitating positive knowledge transfer, leading to performance improvements, particularly in low-resource languages.

For high-resource languages, MMLoRA achieved results similar to those of other PEFT methods, mainly because the language-specific features of these languages had already been sufficiently learned during the pretraining stage of the base models. The one exception is Java on StarCoderBase, where improvements remain observable for two reasons. First, MMLoRA leverages its expert design to capture both common and language-specific features, which benefits all languages, including high-resource ones such as Java. Second, the extent of pretraining varies across models. Java receives extensive pretraining in DeepSeekCoder but considerably less in StarCoderBase, leading to more pronounced gains in the latter.

We also compared the performance of the 1B model fine-tuned with MMLoRA to directly using GPT-4 and Deepseek-v3. Under both one-shot and few-shot settings, MMLoRA achieved Bleu and Meteor scores that were comparable to or exceeded those of the large language models. The results show that without fine-tuning, even strong LLMs struggle to perform well. Besides, our focus is on 1B parameter models, which are better suited for resource-limited environments but often cannot produce well-formatted outputs with prompt engineering alone.

B. RQ2: How do expert selection strategies affect the performance of MMLoRA?

To analyze the impact of expert selection strategies on MMLoRA's performance, we conducted ablation experiments on CodeSearchNet, as all MMLoRA modules are designed around how to select experts. Specifically, we analyzed the results of removing the universal expert, replacing the Global Language Router with a layer-specific Token Router, and removing components of the expert loss function. Additionally, we examined how the ratio between diversity loss and balanced loss affects results in the code summarization task.

Table IV presents the results of ablation studies for MMLoRA. First, we observed that removing the universal expert led to performance declines, especially in low-resource languages like Ruby. Specifically, Bleu and Meteor scores dropped by 15.9% and 10.4%, respectively. This decline highlights the universal expert's role in learning common features shared across languages, which aids in cross-linguistic knowledge transfer and thereby enhances performance for low-resource languages with limited data. Additionally, replacing a single Global Language Router with individual token routers at each layer also reduced performance. Since many programming languages share identical tokens after tokenization, the token router struggles to differentiate these tokens' distinct meanings across various programming contexts, leading to reduced effectiveness. Finally, the ablation studies on the expert loss function suggest that using only the balanced loss or diversity loss alone fails to achieve optimal results. Instead, maintaining a well-calibrated balance between the two

TABLE IV
ABLATION EXPERIMENTAL RESULTS OF OUR METHOD ON DEEPSEEKCODER FOR CODE SUMMARIZATION

Data Availability		High-resource ⟶										Low-resource		Overall	
Ablation Experiments	Param(%)	Java		Python		PHP		Go		Javascript		Ruby			
		Bleu	Meteor	Bleu	Meteor	Bleu	Meteor	Bleu	Meteor	Bleu	Meteor	Bleu	Meteor	Bleu	Meteor
MMLoRA	1.65%	21.38	35.65	**20.95**	**34.13**	**26.83**	**38.08**	21.75	**38.25**	13.80	20.96	**16.55**	**25.65**	**20.19**	**32.12**
w/o universal expert	1.11%	21.08	35.03	20.52	33.31	25.31	36.26	21.38	38.08	13.36	20.08	14.27	23.24	19.32	31.00
w/o Global Language Router	1.65%	20.10	33.73	20.11	33.02	25.52	36.24	21.68	37.19	13.80	20.88	15.46	23.63	19.44	30.78
w/o L_d and L_b	1.65%	**21.85**	**35.68**	20.08	32.45	26.61	36.74	21.28	37.00	13.77	20.71	15.44	23.16	19.84	30.96
w/o L_b	1.65%	21.21	35.00	20.40	33.39	26.74	36.80	21.66	37.68	13.73	**21.13**	15.55	23.08	19.88	31.18
w/o L_d	1.65%	21.45	34.90	20.55	33.72	26.39	36.63	**22.27**	38.22	**13.92**	20.55	15.41	24.00	20.00	31.33

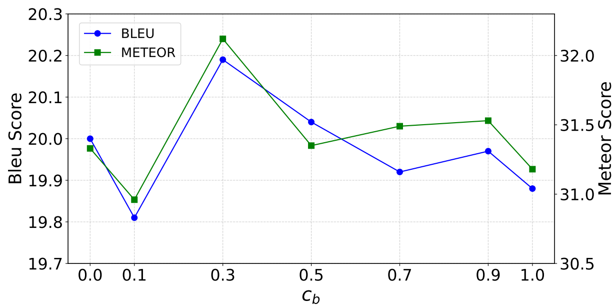


Fig. 4. Evaluation of the performance of MMLoRA when using different weights for diversity loss and a balanced loss on code summarization. Horizontal axis shows the weight of the balanced loss c_b, while the diversity loss c_d is set to $1 - c_b$.

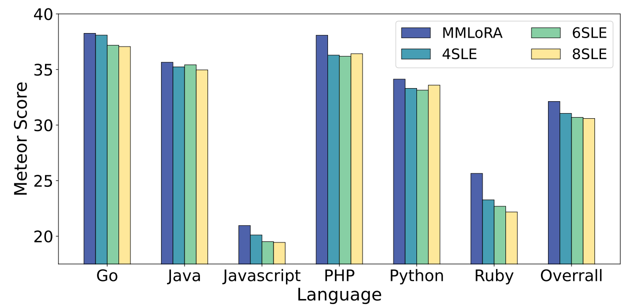


Fig. 5. The performance evaluation on CodeSearchNet dataset using only specialized linguistic experts. *4SLE*, *6SLE*, *8SLE* denote configurations that only use 4, 6, and 8 specialized linguistic experts.

is essential to ensure steady learning progress while preserving the specialization of linguistic experts.

The experimental results in Fig. 4 illustrate the performance of MMLoRA on code summarization using various weights for diversity loss and balanced loss. We tested balanced loss weights c_b range from 0.0 to 1.0, with diversity loss weights c_d set to $1 - c_b$. The results indicate that optimal performance was achieved when $c_d = 0.3$ and $c_b = 0.7$. Balancing diversity loss and balanced loss benefits specialized linguistic experts, leading to more effective learning of both common and language-specific features.

C. RQ3: How does the number of experts affect the performance of MMLoRA?

To further investigate the impact of the number of experts on the final performance of MMLoRA, we examined the total number of specialized linguistic experts and the number selected based on the Top-K criterion. For experimental consistency, MMLoRA was fine-tuned again under each configuration. Specifically, we conducted experiments with 4 and 6 specialized linguistic experts, setting Global Top-K to select 1 expert, 50% of the experts, and 100% of the experts on CodeSearchNet dataset.

Table V presents the experimental results under a varying number of specialized linguistic experts. We observed that increasing the number of specialized linguistic experts from 4 to 6 did not result in noticeable performance improvements, even with an increase in training parameters. This finding aligns with previous results from other MoE research [34]. Additionally, setting Top-K to 1 causes a significant drop in performance, while setting Top-K to the total number of experts does not yield better results. Activating all specialized linguistic experts for each training instance may prevent the experts from focusing on language-specific features.

Moreover, we conducted experiments to investigate whether increasing the number of specialized linguistic experts could serve as a replacement for the universal expert. The results presented in Fig. 5 show that increasing the number of specialized linguistic experts to 6 or 8 still led to a performance decline, suggesting that using only specialized linguistic experts cannot replace the universal expert. The universal expert is essential for capturing common features shared across languages.

D. RQ4: How effective is MMLoRA in low-resource scenarios?

Low-resource scenarios refer to situations where training data is limited. Evaluating performance under low-resource conditions is critical for real-world deployment. Previous approaches [1], [6], [20] have investigated the effectiveness of code summarization with limited data. To evaluate MMLoRA's effectiveness in low-resource scenarios, we compared MMLoRA with LoRA-Mix using reduced code summarization datasets. Specifically, we selected subsets representing 0.1%, 0.5%, and 1% of the original CodeSearchNet dataset to assess code summarization performance under low-resource conditions. We preserved the original inter-language data proportions across different programming languages while creating these subsets.

Fig. 6 presents our experimental results. In addition to the overall performance, we report outcomes for the lowest-

TABLE V
COMPARISON OF EXPERT COUNTS ON DEEPSEEKCODER FOR CODE SUMMARIZATION

Data Availability			High-resource ⟶ Low-resource												Overall	
Experts	Top-K	Param(%)	Java		Python		PHP		Go		Javascript		Ruby			
			Bleu	Meteor	Bleu	Meteor	Bleu	Meteor	Bleu	Meteor	Bleu	Meteor	Bleu	Meteor	Bleu	Meteor
4	1	1.65%	20.29	34.19	20.07	32.34	25.84	36.82	**22.05**	37.25	13.79	21.22	15.29	23.01	19.55	30.80
	2 (50% Experts)	1.65%	21.38	35.65	**20.95**	34.13	**26.83**	**38.08**	21.75	38.25	13.80	20.96	**16.55**	25.65	**20.19**	32.12
	4 (100% Experts)	1.65%	21.85	35.68	20.08	32.45	26.61	36.74	21.28	37.00	13.77	20.71	15.44	23.16	19.84	30.96
6	1	2.47%	20.79	34.87	20.36	32.53	25.61	35.71	21.58	36.78	**14.21**	20.18	15.04	23.00	19.60	30.51
	3 (50% Experts)	2.47%	21.91	36.21	20.81	**34.24**	25.97	37.51	20.81	**38.56**	13.54	**21.76**	15.99	**25.69**	19.92	**32.33**
	6 (100% Experts)	2.47%	**22.59**	**36.38**	19.95	33.77	25.24	35.58	21.84	38.18	13.72	20.40	15.62	23.69	19.83	31.30

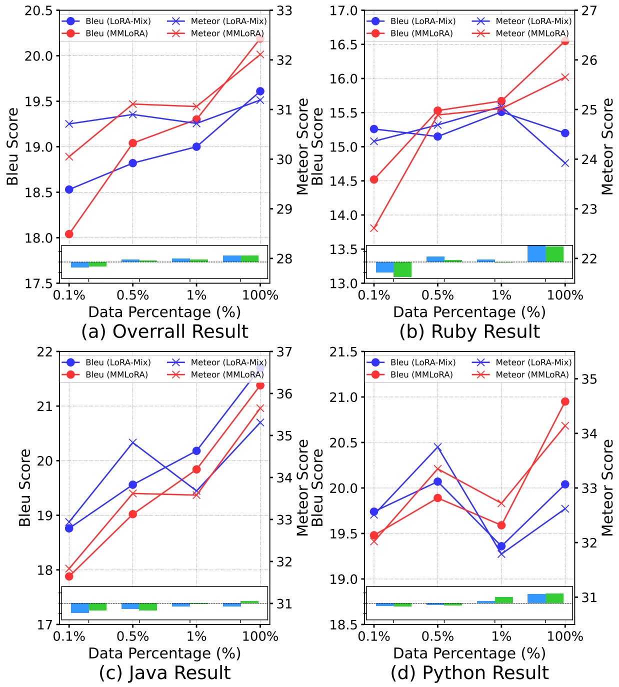


Fig. 6. Evaluation of performance on code summarization datasets of different scales. The bar charts below each subplot illustrate the extent to which MMLoRA improved or declined compared to LoRA-Mix. The blue bars represent changes in Bleu, while the green bars represent changes in Meteor.

resource language Ruby and two high-resource languages Python and Java. When training on only 0.1% of the data, MMLoRA is outperformed by LoRA-Mix. However, as the volume of training data increases, MMLoRA's overall performance improves consistently. This finding suggests that MMLoRA's routing strategy requires a minimum data threshold to effectively activate and select the appropriate specialized linguistic experts.

For the lowest resource language Ruby, which accounts for only 2.7% of the code summarization fine-tuning data, increasing the data volume yields minimal performance gains with LoRA-Mix. This finding indicates that LoRA-Mix does not effectively transfer knowledge from higher-resource languages to low-resource languages during training. In contrast, MMLoRA's performance on Ruby improves significantly as the training data increases. For high-resource languages Java and Python, MMLoRA and LoRA-Mix achieve comparable results, which suggests that MMLoRA can enhance low-resource language performance without diminishing performance for high-resource languages.

TABLE VI
RESULTS OF HUMAN EVALUATION FOR CODE SUMMARIZATION

Methods	Similarity	Naturalness	Informativeness	Relevance	Avg.
GPT-4(one-shot)	1.72	2.28	**2.78**	2.47	2.31
GPT-4(few-shot)	1.87	2.35	2.37	2.41	2.25
FPFT-Mix	2.05	2.33	1.92	2.14	2.17
LoRA-Each	2.25	2.46	2.22	2.48	2.35
LoRA-Mix	2.23	2.44	2.28	2.52	2.37
MoLA	2.20	2.41	2.18	2.46	2.31
MMLoRA	**2.30**	**2.50**	2.44	**2.66**	**2.48**

VI. DISCUSSION

A. *Human Evaluation*

Prior studies [8], [9], [11], [57] have shown that relying solely on automatic evaluation is insufficient, since these metrics may not reliably reflect human judgment. Therefore, we conduct a human evaluation by following the previous works [7]–[11] to assess the summaries generated by five fine-tuning methods on DeepSeekCoder as well as by GPT-4 under one-shot and few-shot prompting settings.

We randomly selected 20 samples for each language in CodeSearchNet, resulting in 120 summary instances for human evaluation. Ten volunteers, each with over three years of programming experience, were recruited to perform the evaluation. Each summary was rated on a scale from 0 to 4 across four dimensions , which is widely used by previous works [7], [8], [10], [11]: **Similarity** (with reference summaries), **Naturalness** (grammatical correctness and fluency), **Informativeness** (amount of relevant information transferred from the input code), and **Relevance** (alignment with the input code snippets).

The results of the human evaluation are presented in Table VI. The standard deviations of all techniques are below 0.6, indicating that the scores assigned by human evaluators exhibit a similar level of consistency. The evaluation results demonstrate that summarization generated by MMLoRA outperform those produced by other fine-tuning methods, with significant improvements observed in terms of Informativeness and Relevance. Moreover, when compared with GPT-4, MMLoRA still demonstrates overall superior performance. This suggests that MMLoRA more effectively learns the language-specific

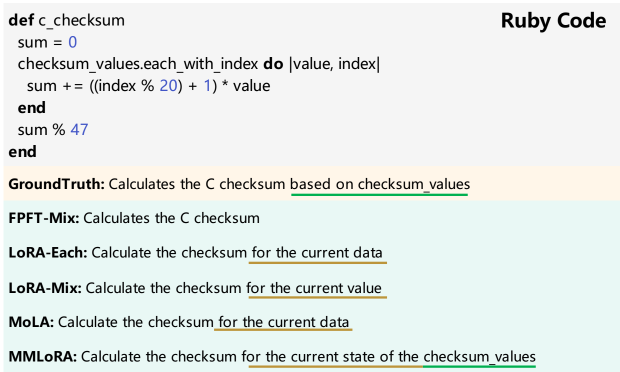


Fig. 7. Examples demonstrating the effectiveness of MMLoRA compared to other methods on a low-resource language Ruby code summarization case.

features of the code itself, thereby generating summaries that are more relevant to the source code and contain more comprehensive information.

B. Case Analysis

To visually demonstrate the effectiveness of MMLoRA, we present an example of Ruby code summarization generated by DeepSeekCoder, as illustrated in Fig. 7. The code calculates the checksum based on the *checksum_values* defined within the function.

Accurately summarizing this Ruby function requires capturing three key elements. First, identifying that the function's primary objective is to calculate a checksum. Second, indicating that the checksum is computed based on a specific variable. Third, explicitly naming the variable *checksum_values*. MMLoRA's output incorporates all these elements while maintaining fluency, generating a summary that is precise and easy to comprehend. In contrast, other PEFT methods (LoRA-Each, LoRA-Mix, MOLA) produce summaries that omit the mention of the variable *checksum_values*, indicating that these methods fail to capture detailed implementation information of low-resource Ruby language. Moreover, the output from FPFT-Mix merely mentions the function's purpose in general terms, which may be attributed to the loss of world knowledge during the full-parameter training process.

C. Expert Allocation

In this section, we visualize the allocation of specialized linguistic experts for code summarization using two base models: DeepSeekCoder and StarCoderBase. The number of specialized linguistic experts is set to 4, with the top-2 experts selected based on the Global Top-K selection strategy. We present the weight distribution of experts for each programming language and the total workload handled by each expert.

Fig. 8 presents heatmaps illustrating the allocation of specialized linguistic experts. We find that different models result in different allocations of experts. The overall weights of all experts are similar on both models, which can be attributed to the expert loss function that facilitates the balance among specialized linguistic experts. For low-resource languages, the

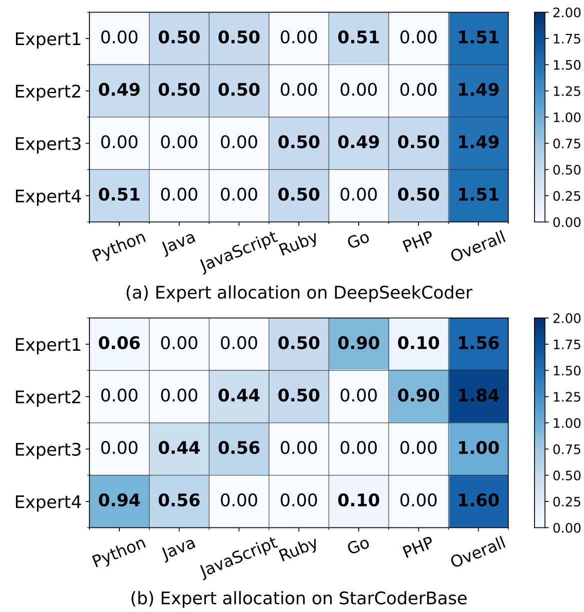


Fig. 8. Specialized linguistic experts allocation of code summarization on DeepSeekCoder and StarCoderBase.

weights of the two selected specialized linguistic experts are more similar, which facilitates knowledge transfer. Moreover, linguistically similar languages tend to share experts, Ruby and PHP share some language-specific experts in both models, due to their common nature as interpreted languages widely used in web development. Specifically, the distribution of experts is relatively balanced in DeepSeekCoder, with two specialized experts assigned to each language having nearly equal weights. For StarCoderBase, high-resource languages such as Python, PHP, and Go more rely on a single specialized linguistic expert. In contrast, the weights of the two selected experts remain fairly similar for low-resource languages. The results showing that MMLoRA automatically determines how to share experts for better transfer.

VII. Related Work

A. Code Summarization with Deep Learning

Deep learning models have advanced the state-of-the-art in software engineering tasks, such as code summarization. Early work used Recurrent Neural Network (RNN) [58], [59] to implement sequence-to-sequence models for code summarization. Some studies have incorporated tree structures [59]–[61] or graph structures [62] to learn detailed code structural information. With the development of Transformer [63], pre-trained models have achieved tremendous success, which has also led to the emergence of numerous CLMs. CLMs can be categorized into three types: encoder-only (e.g. CodeBERT [3] and GraphCodeBERT [4]), decoder-only (e.g. CodeGen [17], StarCoder [18], and DeepSeekCoder [19]), and encoder-decoder (e.g. CodeT5 [64], GrammarT5 [65] and UniXCoder [66]), all of which have shown strong performance in code summarization. Recent studies have improved the code summarization performance of CLMs through model interpretation [67] or prompt engineering [68], [69].

However, when performing code summarization for programming languages across multiple domains or languages, it is often necessary to fine-tune the CLMs to adapt them to downstream tasks. Fine-tuning all parameters is both time-consuming and resource-intensive. Therefore, various approaches have been proposed, such as training on a mixture of languages [1], [20], [26] or using PEFT methods [20], [23], [70] to fine-tune. Liu et al. [23] evaluated different fine-tuning methods on CodeT5 [64] and PLBART [71], evaluating their performance across multiple software engineering tasks including code summarization. Ahmed et al. [1] find that code written in different languages often shares similar variable names, which are important indicators for code summarization. They demonstrate through experiments that training on a multilingual mixture improves performance in both CodeBERT [3] and GraphCodeBERT [4]. Wang et al. [20] combined all languages and employed adapter fine-tuning, effectively reducing computational resource consumption while avoiding catastrophic forgetting of world knowledge caused by FPFT.

B. Low-resource Language in Software Engineering

Low-resource languages are programming languages with insufficient training data. Many studies focus on improving or analyzing the performance of software engineering tasks for low-resource languages, such as code summarization [1], [6], [20], [72], code generation [73], [74], code repair [75], [76], and clone detection [77]. Specifically, Chen et al. [6] investigated the transferability of CLMs for low-resource programming languages in the code summarization task and proposed effective strategies for selecting suitable programming languages for fine-tuning. SPEAC [73] is a novel approach that enables CLMs to generate syntactically valid code in very low-resource programming languages by introducing an intermediate language and leveraging compiler techniques. SELF-DEBUGGING [75] enables CLMs to iteratively debug their own code generation without human feedback. Other works [74], [78] translate popular pre-training datasets and monolingual benchmarks into a diverse range of programming languages, including many low-resource languages. Previous studies [6], [73], [73], [75] have focused on mitigating the performance degradation of low-resource languages in a given multilingual task. In contrast, MMLoRA introduces a novel fine-tuning approach that not only retains common features shared across multiple languages but also effectively mitigates gradient conflicts, thereby improving the performance of low-resource languages in downstream tasks.

VIII. Threats to Validity

External Validity concerns the generalizability of study findings to other settings and datasets. We evaluated the effectiveness of MMLoRA and other fine-tuning methods using two base models on code summarization task across six programming languages. Previous studies [79] suggested that code summarization datasets may suffer from inconsistent annotations. To enhance the reliability of our findings, we used the widely adopted CodeSearchNet and XLCoST datasets, supplemented by human evaluation. However, we did not include certain promising PEFT methods such as Adapter Tuning [41], or extend our comparison of PEFT methods across more base models, which could impact the generalizability of our findings. Besides, applying LoRA [22] at different positions affects both the proportion of trainable parameters and the overall model performance. To ensure fairness in comparison, all LoRA-based PEFT methods were consistently applied to the FFN layers and Attention layers in both models.

Internal Validity concerns unanticipated relationships that could impact the study's results. Although MMLoRA significantly reduces the number of trainable parameters compared to FPFT methods, thereby avoiding resource-intensive issues. The expert selection process which relies on the router's output still results in longer evaluation and training times. To mitigate this issue. Unlike other methods [32], [33] that add a router to each multi-expert module, we employ a single Global Language Router to achieve relatively efficient global expert selection. However, the Global Language Router leads to the current MMLoRA being unable to effectively generalize to untrained languages. We will explore other routing strategies to make MMLoRA more generalizable in the future.

IX. Conclusion And Future Work

In this paper, we introduce Mixture-of-Experts Multilingual Low-Rank Adaptation, which is a novel PEFT approach for CLMs in software engineering tasks. MMLoRA updates less than 2% of the model's parameters, significantly reducing computational overhead compared to FPFT. MMLoRA extends the MoE structure by utilizing a universal expert and a set of specialized linguistic experts to retain common features while capturing language-specific features to mitigate gradient conflicts. Experimental results show that MMLoRA achieves state-of-the-art performance in code summarization, particularly for low-resource languages. In future work, we plan to evaluate the effectiveness of MMLoRA on a broader range of base models and explore more efficient routing strategies. Additionally, we aim to expand MMLoRA across a broader range of multilingual other code-related tasks, such as code translation and code repair to validate its generalizability. Our code and experimental data are publicly available at https://github.com/UnbSky/MMLoRA.

X. Acknowledgement

This research is supported by the National Natural Science Foundation of China (62476097, 62402185), the Fundamental Research Funds for the Central Universities, South China University of Technology (x2rjD2250190), Guangdong Provincial Fund for Basic and Applied Basic Research—Regional Joint Fund Project (Key Project) (2023B1515120078), Guangdong Provincial Natural Science Foundation for Outstanding Youth Team Project (2024B1515040010), Basic and Applied Basic Research (Young Doctor "QiHang" Project)—Science and Technology Projects of Guangzhou (SL2023A04J00753).

References

[1] T. Ahmed and P. Devanbu, "Multilingual training for software engineering," in *Proceedings of the ICSE 2022*, 2022, pp. 1443–1455.

[2] M. D. Ernst, "Natural language is a programming language: Applying natural language processing to software development," in *Proceedings of the SNAPL 2017*, 2017.

[3] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," in *Findings of the EMNLP 2020*, 2020, pp. 1536–1547.

[4] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, L. Shujie, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," in *Proceedings of the ICLR 2020*, 2020.

[5] A. Mastropaolo, M. Ciniselli, L. Pascarella, R. Tufano, E. Aghajani, and G. Bavota, "Towards summarizing code snippets using pre-trained transformers," in *Proceedings of the ICPC 2024*, 2024, pp. 1–12.

[6] 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 ICPC 2022*, 2022, pp. 401–412.

[7] 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.

[8] 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 ICSE 2022*, 2022, pp. 1597–1608.

[9] 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.

[10] W. Sun, Y. Miao, Y. Li, H. Zhang, C. Fang, Y. Liu, G. Deng, Y. Liu, and Z. Chen, "Source code summarization in the era of large language models," in *Proceedings of the ICSE 2024*. IEEE Computer Society, 2024, pp. 419–431.

[11] R. Haldar and J. Hockenmaier, "Analyzing the performance of large language models on code summarization," in *Proceedings of LREC-COLING 2024)*, 2024, pp. 995–1008.

[12] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *Proceedings of the ICSE 2019*. IEEE, 2019, pp. 795–806.

[13] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the ASE 2018*, 2018, pp. 397–407.

[14] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of the ICML 2016*. PMLR, 2016, pp. 2091–2100.

[15] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *Proceedings of the ICLR 2019*, 2019.

[16] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: exemplar-based neural comment generation," in *Proceedings of the ICSE 2020*, 2020, pp. 349–360.

[17] 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," in *Proceedings of the ICLR 2023*, 2023.

[18] R. Li, L. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: May the source be with you!" *Transactions on machine learning research*, 2023.

[19] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming - the rise of code intelligence," *ArXiv*, vol. abs/2401.14196, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:267211867

[20] 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 *Proceedings of the ICSE 2023*, 2023, pp. 5–16.

[21] X. L. Li and P. Liang, "Prefix-tuning: Optimizing continuous prompts for generation," in *Proceedings of the ACL 2021*, 2021, pp. 4582–4597.

[22] E. J. Hu, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen *et al.*, "Lora: Low-rank adaptation of large language models," in *Proceedings of the ICLR 2022*, 2021.

[23] J. Liu, C. Sha, and X. Peng, "An empirical study of parameter-efficient fine-tuning methods for pre-trained code models," in *Proceedings of the ASE 2023*, 2023, pp. 397–408.

[24] A. Silva, S. Fang, and M. Monperrus, "Repairllama: Efficient representations and fine-tuned adapters for program repair," *arXiv preprint arXiv:2312.15698*, 2023.

[25] 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.

[26] S. Lakew, M. Cettolo, and M. Federico, "A comparison of transformer and recurrent neural networks on multilingual neural machine translation," in *Proceedings of the COLING 2018*, 2018, pp. 641–652.

[27] X. Tan, Y. Ren, D. He, T. Qin, Z. Zhao, and T.-Y. Liu, "Multilingual neural machine translation with knowledge distillation," in *Proceedings of the ICLR 2018*, 2018.

[28] Y. Zhang and Q. Yang, "A survey on multi-task learning," *IEEE transactions on knowledge and data engineering*, vol. 34, no. 12, pp. 5586–5609, 2021.

[29] Z. Yin, J. Wang, J. Cao, Z. Shi, D. Liu, M. Li, X. Huang, Z. Wang, L. Sheng, L. Bai *et al.*, "Lamm: Language-assisted multi-modal instruction-tuning dataset, framework, and benchmark," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[30] B. Liu, X. Liu, X. Jin, P. Stone, and Q. Liu, "Conflict-averse gradient descent for multi-task learning," *Proceedings of the NeurIPS 2021*, vol. 34, pp. 18 878–18 890, 2021.

[31] J. Chen and M. J. Er, "Mitigating gradient conflicts via expert squads in multi-task learning," *Neurocomputing*, vol. 614, p. 128832, 2025.

[32] Q. Liu, X. Wu, X. Zhao, Y. Zhu, D. Xu, F. Tian, and Y. Zheng, "When moe meets llms: Parameter efficient fine-tuning for multi-task medical applications," in *Proceedings of the ACM SIGIR 2024*, 2024, pp. 1104–1114.

[33] W. Feng, C. Hao, Y. Zhang, Y. Han, and H. Wang, "Mixture-of-loras: An efficient multitask tuning for large language models," in *Proceedings of the COLING 2024*, 2024, pp. 11 371–11 380.

[34] S. Dou, E. Zhou, Y. Liu, S. Gao, W. Shen, L. Xiong, Y. Zhou, X. Wang, Z. Xi, X. Fan *et al.*, "Loramoe: Alleviating world knowledge forgetting in large language models via moe-style plugin," in *Proceedings of the ACL 2024*, 2024, pp. 1932–1945.

[35] Y. Gou, Z. Liu, K. Chen, L. Hong, H. Xu, A. Li, D.-Y. Yeung, J. T. Kwok, and Y. Zhang, "Mixture of cluster-conditional lora experts for vision-language instruction tuning," *arXiv preprint arXiv:2312.12379*, 2023.

[36] D. Li, Y. Ma, N. Wang, Z. Cheng, L. Duan, J. Zuo, C. Yang, and M. Tang, "Mixlora: Enhancing large language models fine-tuning with lora based mixture of experts," *arXiv preprint arXiv:2404.15159*, 2024.

[37] T. Luo, J. Lei, F. Lei, W. Liu, S. He, J. Zhao, and K. Liu, "Moelora: Contrastive learning guided mixture of experts on parameter-efficient fine-tuning for large language models," *arXiv preprint arXiv:2402.12851*, 2024.

[38] L. Yuan, Y. Cai, X. Shen, Q. Li, Q. Huang, Z. Deng, and T. Wang, "Collaborative multi-lora experts with achievement-based multi-tasks loss for unified multimodal information extraction," in *Proceedings of the IJCAI 2025*, 2025, pp. 6940–6948.

[39] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.

[40] M. Zhu, A. Jain, K. Suresh, R. Ravindran, S. Tipirneni, and C. K. Reddy, "Xlcost: A benchmark dataset for cross-lingual code intelligence," 2022. [Online]. Available: https://arxiv.org/abs/2206.08474

[41] B. Lester, R. Al-Rfou, and N. Constant, "The power of scale for parameter-efficient prompt tuning," in *Proceedings of the EMNLP 2021*, 2021, pp. 3045–3059.

[42] 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 *Proceedings of the ICLR 2019*, 2019, pp. 2790–2799.

[43] S.-y. Liu, C.-Y. Wang, H. Yin, P. Molchanov, Y.-C. F. Wang, K.-T. Cheng, and M.-H. Chen, "Dora: Weight-decomposed low-rank adaptation," in *Proceedings of the ICML 2024*, 2024.

[44] A. Renduchintala, T. Konuk, and O. Kuchaiev, "Tied-lora: Enhancing parameter efficiency of lora with weight tying," in *Proceedings of the NAACL 2024*, 2024, pp. 8686–8697.

[45] Q. Zhang, M. Chen, A. Bukharin, P. He, Y. Cheng, W. Chen, and T. Zhao, "Adaptive budget allocation for parameter-efficient fine-tuning," in *Proceedings of the ICLR 2023*, 2023.

[46] J. Kumar and S. Chimalakonda, "Code summarization without direct access to code-towards exploring federated llms for software engineering," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2024, pp. 100–109.

[47] C.-Y. Su and C. McMillan, "Semantic similarity loss for neural source code summarization," *Journal of Software: Evolution and Process*, p. 2706, 2023.

[48] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton, "Adaptive mixtures of local experts," *Neural computation*, vol. 3, no. 1, pp. 79–87, 1991.

[49] C. Gao, K. Chen, J. Rao, B. Sun, R. Liu, D. Peng, Y. Zhang, X. Guo, J. Yang, and V. Subrahmanian, "Higher layers need more lora experts," *arXiv preprint arXiv:2402.08562*, 2024.

[50] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *Journal of Machine Learning Research*, vol. 23, no. 120, pp. 1–39, 2022.

[51] 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," in *Proceedings of the NeurIPS 2021*, 2021.

[52] S. Zhang, L. Shuiyan, Q. Rongzhi, and X. Zhou, "Codefuse: Multimodal code search model with fine-grained attention alignment," in *Proceedings of the 2024 IEEE 48th Annual Computers, Software, and Applications Conference*. IEEE, 2024, pp. 1290–1299.

[53] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the ACL 2002*, 2002, pp. 311–318.

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

[55] X. Hu, Q. Chen, H. Wang, X. Xia, D. Lo, and T. Zimmermann, "Correlating automated and human evaluation of code documentation generation quality," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 4, pp. 1–28, 2022.

[56] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Evaluating large language models in class-level code generation," in *Proceedings of the ICSE 2024*, 2024, pp. 1–13.

[57] 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 ICSE 2024*, 2024, pp. 1–13.

[58] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the ACL 2016*, 2016, pp. 2073–2083.

[59] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the ICPC 2018*, 2018, pp. 200–210.

[60] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proceedings of the ICSE 2019*, 2019, pp. 783–794.

[61] Z. Yang, J. Keung, X. Yu, X. Gu, Z. Wei, X. Ma, and M. Zhang, "A multi-modal transformer-based code summarization approach for smart contracts," in *Proceedings of the ICPC 2021*, 2021, pp. 1–12.

[62] Y. Wang, Y. Dong, X. Lu, and A. Zhou, "Gypsum: learning hybrid representations for code summarization," in *Proceedings of the ICPC 2022*, 2022, pp. 12–23.

[63] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[64] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proceedings of the EMNLP 2021*, 2021, pp. 8696–8708.

[65] Q. Zhu, Q. Liang, Z. Sun, Y. Xiong, L. Zhang, and S. Cheng, "Grammart5: Grammar-integrated pretrained encoder-decoder neural model for code," in *Proceedings of the ICSE 2024*, 2024, pp. 1–13.

[66] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the ACL 2022*, 2022, pp. 7212–7225.

[67] M. Geng, S. Wang, D. Dong, H. Wang, S. Cao, K. Zhang, and Z. Jin, "Interpretation-based code summarization," in *Proceedings of the ICPC 2023*, 2023, pp. 113–124.

[68] T. Xu, Y. Miao, C. Fang, H. Qian, X. Feng, Z. Chen, C. Wang, J. Zhang, W. Sun, Z. Chen, and Y. Liu, "A prompt learning framework for source code summarization," *arXiv preprint arXiv:2312.16066*, 2024.

[69] M. Fang, X. Yuan, Y. Li, H. Li, C. Fang, and J. Du, "Enhanced prompting framework for code summarization with large language models," *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3728949

[70] 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.

[71] W. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "Unified pre-training for program understanding and generation." in *Proceedings of the NAACL 2021*, 2021.

[72] T. Ahmed and P. Devanbu, "Few-shot training llms for project-specific code-summarization," in *Proceedings of the ASE 2022*, 2022, pp. 1–5.

[73] F. Mora, J. Wong, H. Lepe, S. Bhatia, K. Elmaaroufi, G. Varghese, J. E. Gonzalez, E. Polgreen, and S. Seshia, "Synthetic programming elicitation for text-to-code in very low-resource programming and formal languages," *Proceedings of the NeurIPS 2024*, vol. 37, pp. 105 151–105 170, 2024.

[74] F. Cassano, J. Gouwar, F. Lucchetti, C. Schlesinger, A. Freeman, C. J. Anderson, M. Q. Feldman, M. Greenberg, A. Jangda, and A. Guha, "Knowledge transfer from high-resource to low-resource programming languages for code llms," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 677–708, 2024.

[75] X. Chen, M. Lin, N. Schaerli, and D. Zhou, "Teaching large language models to self-debug," in *Proceedings of the ACL 2023*, 2023.

[76] K. Wong, A. Amayuelas, L. Pan, and W. Y. Wang, "Investigating the transferability of code repair for low-resource programming languages," *arXiv preprint arXiv:2406.14867*, 2024.

[77] R. Baltaji, S. Pujar, L. Mandel, M. Hirzel, L. Buratti, and L. Varshney, "Learning transfers over several programming languages," *arXiv preprint arXiv:2310.16937*, 2023.

[78] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman *et al.*, "Multipl-e: A scalable and extensible approach to benchmarking neural code generation," *arXiv preprint arXiv:2208.08227*, 2022.

[79] A. Vitale, A. Mastropaolo, R. Oliveto, M. Di Penta, and S. Scalabrino, "Optimizing datasets for code summarization: Is code-comment coherence enough?" *arXiv preprint arXiv:2502.07611*, 2025.