# Mix-of-Language-Experts Architecture for Multilingual Programming

Yifan Zong
*University of Waterloo*
Waterloo, Canada
y22zong@uwaterloo.ca

Yuntian Deng
*University of Waterloo*
Waterloo, Canada
yuntian@uwaterloo.ca

Pengyu Nie
*University of Waterloo*
Waterloo, Canada
pynie@uwaterloo.ca

***Abstract*—Large language models (LLMs) have demonstrated impressive capabilities in aiding developers with tasks like code comprehension, generation, and translation. Supporting multilingual programming—i.e., coding tasks across multiple programming languages—typically requires either (1) finetuning a single LLM across all programming languages, which is cost-efficient but sacrifices language-specific specialization and performance, or (2) finetuning separate LLMs for each programming language, which allows for specialization but is computationally expensive and storage-intensive due to the duplication of parameters.**

**This paper introduces MoLE (Mix-of-Language-Experts), a novel architecture that balances efficiency and specialization for multilingual programming. MoLE is composed of a base model, a shared LoRA (low-rank adaptation) module, and a collection of language-specific LoRA modules. These modules are jointly optimized during the finetuning process, enabling effective knowledge sharing and specialization across programming languages. During inference, MoLE automatically routes to the language-specific LoRA module corresponding to the programming language of the code token being generated. Our experiments demonstrate that MoLE achieves greater parameter efficiency compared to training separate language-specific LoRAs, while outperforming a single shared LLM finetuned for all programming languages in terms of accuracy.**

***Index Terms*—large language model for code, multilingual programming, low-rank adaptation, mix of experts**

## I. Introduction

Large Language Models (LLMs) have transformed software development by enabling advanced code comprehension, generation, and translation capabilities [9], [15], [25], [31], [34], [35]. Trained on massive code repositories, LLMs can enhance developer productivity by automating routine tasks and providing intelligent code completions. The success of tools like GitHub Copilot [5], Amazon CodeWhisperer [1], and Cursor [3] demonstrates the transformative impact of LLMs in software development.

To support multilingual programming—coding tasks across multiple programming languages—current LLMs are typically pretrained and finetuned on diverse multilingual code repositories. While this generalist approach allows broad applicability, it struggles with language-specific nuances, leading to suboptimal performance for individual languages. To address this, language-specific finetuning has been explored, improving specialization but at significant computational and storage costs. Moreover, independently finetuned models fail to share knowledge across languages, limiting performance in low-resource languages where data scarcity is common.

To bridge this gap, we propose **MoLE** (**M**ix-**o**f-**L**anguage-**E**xperts), a novel architecture that balances efficiency and specialization in multilingual programming. MoLE employs a *shared adapter* (used by programming language tokens) and an *NL adapter* (used by natural language tokens) to store the common knowledge about programming patterns and concepts. Another set of *expert adapters* store language-specific knowledge about each programming language's syntax and semantics. During training, the language-agnostic and language-specific adapters are jointly optimized, enabling effective knowledge sharing while maintaining specialization. At inference, an appropriate language-specific adapter is automatically activated according to each token's language.

Built on low-rank adaptation (LoRA) [17], MoLE efficiently finetunes LLMs by updating only a small subset of parameters, with the frozen pretrained model serving as a foundation of multilingual programming knowledge. Given that the pretrained model already contains multilingual programming knowledge in a less organized manner, the primary goal of MoLE's finetuning is to shift the language-agnostic and language-specific knowledge into the corresponding LoRA adapters. To bootstrap this process, we apply a principal-components-based initialization strategy [27] in MoLE, with the most important components assigned to the shared adapter, followed by the expert adapters.

We evaluated MoLE on a multilingual code assistant dataset spanning eight programming languages and tested it across three tasks: code summarization, synthesis, and translation. MoLE consistently outperforms both the all-language finetuning baseline (a single shared model for all languages) and per-language finetuning baseline (separate models for each language). An analysis of individual expert performance confirms that MoLE effectively organizes language-agnostic and language-specific knowledge into distinct adapter components. The main contributions of this work include:

- **Architecture**. A Mix-of-Language-Experts design that enables parameter-efficient finetuning for multilingual programming while separating language-agnostic and language-specific knowledge.
- **Implementation**. A fully open-source system that jointly finetunes adapters for eight programming languages.

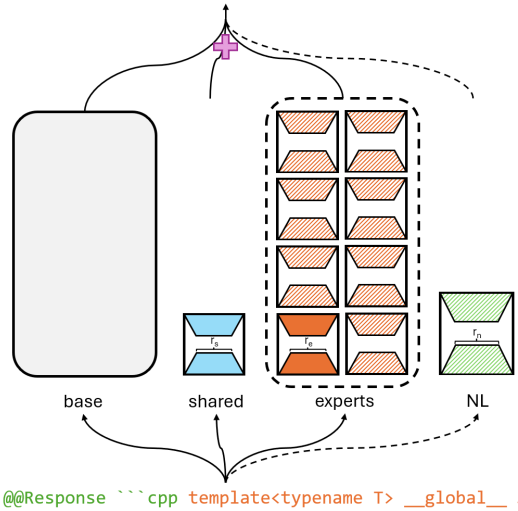


Fig. 1: Architecture of MoLE. The model extends the standard transformer by incorporating three types of LoRA adapters: a shared adapter (capturing commonalities across all programming languages), expert adapters (specializing in language-specific syntax and semantics), and an NL adapter (processing natural language tokens). During finetuning, the adapters are conditionally activated based on the input token's language, enabling efficient knowledge sharing and specialization.

- **Evaluation**. Comprehensive experiments demonstrating MoLE's superior performance on code summarization, synthesis, and translation tasks. We also confirmed the effectiveness of our proposed architecture in disseminating knowledge across multiple programming languages.

MoLE's code and model checkpoints are open-sourced at: https://github.com/uw-swag/mole

## II. MoLE Technique

### A. Architecture

Figure 1 illustrates MoLE's architecture, which extends the standard transformer architecture [33] by incorporating three types of LoRA adapters [17]: a *shared adapter*, *expert adapters*, and a *NL adapter*, in each linear layer of the feed-forward network. The original transformer model serves as the *base model*, while the adapters specialize in processing specific programming or natural languages, following the principle of separation of concerns [22].

LoRA [17] is a parameter-efficient technique for finetuning transformer models under the assumption that parameter updates have a low intrinsic rank. For a linear layer with weight matrix $W_0 \in \mathbb{R}^{d\times k}$, the LoRA adapter introduces two trainable matrices $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$, where $r \ll min(d,k)$ is called the *rank* of the adapter. The weight update is then constrained to a low-rank decomposition $\Delta W = BA$, significantly reducing the number of trainable parameters. For example, in a 1.3B parameter transformer model, a linear layer with $d \times k = 2048 \times 5504 \approx 11.2$M parameters requires only 483k trainable parameters with $r = 64$ (4.3% of the original layer's parameters).

MoLE leverages $K + 2$ LoRA adapters for $K$ programming languages: a shared adapter ($\Delta W_s$), $K$ expert adapters (denoted as $\Delta W_e$), and an NL adapter ($\Delta W_n$). Adapters are conditionally activated based on the token's language:

- For programming language tokens:

$$W = W_0 + \Delta W_s + \Delta W_e = W_0 + B_s A_s + B_e A_e$$

- For natural language tokens:

$$W = W_0 + \Delta W_n = W_0 + B_n A_n$$

All the expert adapters have the same rank, and the rank of the NL adapter is the same as the combined ranks of the shared adapter and one expert adapter, i.e., $r_n = r_s + r_e$.

The shared adapter captures common patterns across languages (e.g., control flow syntax), while expert adapters specialize in language-specific constructs (e.g., Python's `elif` keyword). The NL adapter processes natural language tokens, including user instructions and code explanations.

### B. Finetuning

MoLE finetunes the pretrained base transformer by disentangling its knowledge into adapters specialized for different languages. Unlike the "noise & zero" initialization in standard LoRA [17], MoLE uses a *principal-components-based* initialization adapted from PiSSA [27], leveraging the pretrained model's knowledge. Specifically, for each linear layer with weight $W \in \mathbb{R}^{d\times k}$:

1) Perform singular value decomposition (SVD):

$$W = USV^T.$$

Here, $U \in \mathbb{R}^{d\times \min(d,k)}$ and $V \in \mathbb{R}^{k\times \min(d,k)}$ are singular vectors, and $S \in \mathbb{R}^{\min(d,k)\times \min(d,k)}$ is a diagonal matrix containing the singular values in descending order. Conceptually, the larger singular values are, the more important the corresponding parameter components (represented by the singular vectors) are. Therefore, we assign the most important $r_n$ principal components to the adapters in MoLE.

2) Initialize LoRA adapters with assigned principle components: Since the common knowledge is the basis of the language-specific knowledge, we assign the first $r_s$ principal components to the shared adapter, and the remaining $r_e$ principal components to the expert adapters. The NL adapter, which is on an alternative computation flow, is always assigned all $r_n = r_s + r_e$ principal components. The adapters are initialized as:

$$A_s = U_{[:,:r_s]} S^{1/2}_{[:r_s,:r_s]}, B_s = S^{1/2}_{[:r_s,:r_s]} V^T_{[:,:r_s]}$$
$$A_e = U_{[:,r_s:r_n]} S^{1/2}_{[r_s:r_n,r_s:r_n]}, B_e = S^{1/2}_{[r_s:r_n,r_s:r_n]} V^T_{[:,r_s:r_n]}$$
$$A_n = U_{[:,:r_n]} S^{1/2}_{[:r_n,:r_n]}, B_n = S^{1/2}_{[:r_n,:r_n]} V^T_{[:,:r_n]}$$

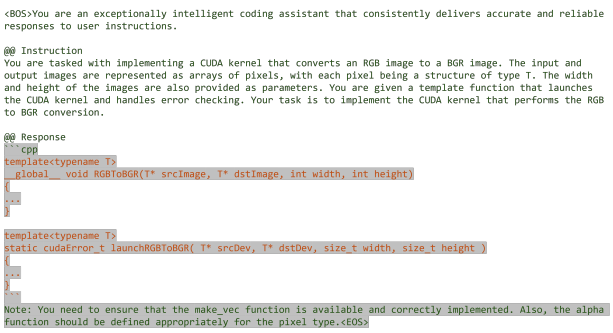


Fig. 2: Example finetuning data. Green tokens represent natural language, while orange tokens represent C++. Grey background indicates the target tokens where loss is computed.

3) Finally, the residual principal components are used as the base model's parameters:

$$W_0 = U_{[:,r_n:]} S_{[r_n:,r_n:]} V^T_{[:,r_n:]}$$

Our principal-components-based initialization strategy preserves the full capability of the pretrained model (i.e., $W = W_0 + B_s A_s + B_e A_e = W_0 + B_n A_n$), and allows each adapter to evolve based on the base model's knowledge during the finetuning process.

Finetuning is performed with the next-token prediction objective on the target tokens, activating adapters based on token language labels. Figure 2 shows an example of the finetuning data, where the tokens belong to two languages: C++ and natural language. Involving multiple languages in the same finetuning data allows for the loss to back-propagate through adapters that belong to different languages, and prompts the rearrangement of knowledge across different adapters.

### C. Inference

During inference, MoLE dynamically activates adapters based on the token's language. Two modes are supported:

- **Automated Mode**. MoLE detects code block boundaries to switch adapters. Specifically, MoLE starts off with using the NL adapter for generating natural language; upon generating the code block opening tokens "```{lang}" (where `lang` is a programming language name, e.g., `python`), MoLE switches to use `lang`'s expert adapter for generating code; and then upon generating the code block closing token "```", MoLE switches back to the NL adapter for generating natural language. This mode is useful for general purpose code assistant tasks, e.g., code generation (with optional natural language explanations) and code summarization. For example, when generating the response in Figure 2, MoLE will automatically switch from natural language to C++ (for the code block) and back to natural language (for the notes in the end).
- **Manual Mode**. The user specifies a target language, activating its corresponding adapter throughout generation. This mode is useful when the output should be entirely in one specific programming language, e.g., code translation to a specific target language. Figure 4 illustrates an example where C++ is used exclusively for translation.

TABLE I: Statistics of the finetuning dataset.

| Language | #Samples |
|---|---|
| C | 15,773 |
| C++ | 37,136 |
| C# | 11,320 |
| Go | 9,533 |
| Java | 24,010 |
| JS | 22,958 |
| Python | 52,642 |
| Rust | 18,835 |
| **Total** | 192,207 |

## III. Dataset

### A. Finetuning Dataset

We choose the Glaive-code-assistant-v3 dataset [6] as the finetuning dataset, because it covers a diverse set of programming languages and is suitable for instruction-tuning general-purpose code assistants. The dataset contains 950K coding problems and solutions generated using Glaive's synthetic data generation platform. We classify dataset samples based on code block annotations and find that, although the dataset covers over 1,000 programming languages, most have few data samples. In this work, we focus on eight frequently-used programming languages with the most data in the Glaive-code-assistant-v3 dataset: C, C++, C#, Go, Java, JS (JavaScript), Python, and Rust. We also observe that some of the problems and solutions are mostly in natural language (e.g., questions asking for third-party library suggestions), which diverge from the goal of this work to assist with programming tasks. Thus, we set a filter that requires at least 1/3 of the characters in a sample (question + answer) belong to code blocks. After filtering, we are left with a high-quality finetuning dataset of more than 192K samples and 7.86M tokens. We randomly sample 5% of data from each programming language as the validation set.

Table I shows the distribution of the number of samples over languages. Python has the most amount of data, which may be due to its popularity in the training and evaluation sets of LLMs for code. The number of samples in all languages are on a similar scale, with the lowest-resource language being Go with 9,533 samples. An example of the finetuning data is shown in Figure 2.

### B. Evaluation Benchmarks

We study the performance of MoLE on three distinct coding tasks: code summarization, synthesis, and translation. To facilitate our evaluation, we build on the HumanEvalPack benchmark [2], [28], which extends the Python-only HumanEval benchmark [9] to six programming languages: C++, Go, Java, JS, Python, and Rust.

We directly adopt HumanEvalPack's "explain code" and "synthesize code" tasks as the first two tasks in our evaluation,

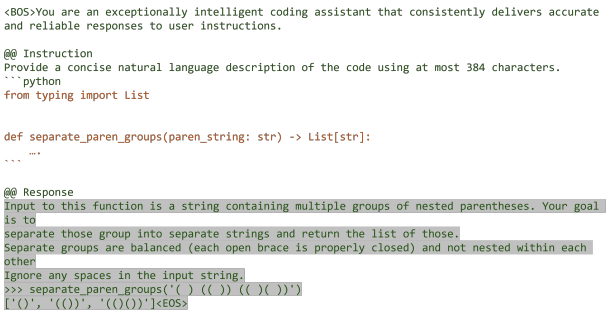


Fig. 3: Example of the summarization task. Green tokens represent natural language, while orange tokens represent Python. Tokens with a grey background indicate the target outputs that the model should generate.

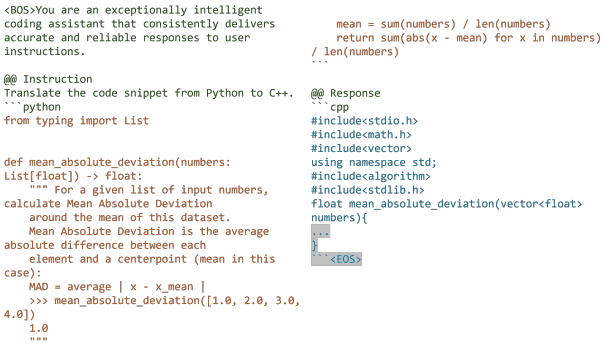


Fig. 4: Example of the translation task. Green tokens represent natural language, orange tokens represent Python, and blue tokens correspond to C++. Tokens with a grey background indicate the target outputs that the model should generate.

summarization and synthesis. Each task contains 164 samples per language, for a total of 984 samples. Figure 3 shows an example of the summarization task, where the model is given a Python code snippet, and asked to generate a natural language summary of the code.

To study the performance of MoLE on the translation task, we leverage the fact that HumanEvalPack was developed by translating HumanEval's each Python sample to the other five languages. Thus, we construct a translation benchmark with all the 30 combinations of {source language, target language} among the six programming languages in HumanEvalPack, for a total of 4,920 samples. Figure 4 shows an example of the translation task, where the model is given a code snippet in Python, and asked to translate the code to C++.

## IV. Evaluation

We aim to address the following research questions:

**RQ1**. How does MoLE perform compared to baseline LoRA finetuning and full finetuning strategies?

**RQ2**. What is the optimal distribution of shared adapter rank versus expert adapter rank in MoLE?

**RQ3**. How do alternative design choices in MoLE, including the initialization approaches of LoRA adapters and the usage of NL adapter, impact its performance?

**RQ4**. Does each expert adapter in MoLE effectively capture language-specific knowledge?

### A. Baselines

We use DeepSeek Coder 1.3B Base [4], [15] as the frozen pretrained model on which we finetune baselines and MoLE. We consider the following baselines:

**No-FT** is the base pretrained model without any finetuning, used as the reference point to evaluate the benefit of finetuning.

**AllLang-FT** is the model finetuned on all programming languages, using LoRA finetuning. The LoRA adapter in this baseline has a rank of 64, to be comparable to MoLE.

**PerLang-FT** is the combination of eight models, where each model is finetuned with the subset of the finetuning dataset that contains only one programming language. Each model is LoRA finetuned with rank of 64. Upon inference, the model for the programming language corresponding to the problem's language will be used (this baseline is thus not applicable for the translation task which involves two languages).

**Full-FT** is the model finetuned on all programming languages, with all parameters trainable. Note that MoLE, as a parameter-efficient finetuning technique, is *not* designed to outperform full finetuning, so we use the full finetuning baseline as the "upper-bound" reference point when comparing MoLE and other parameter-efficient finetuning baselines.

### B. MoLE and Variants

We set the rank of the NL adapter (i.e., the sum of the shared adapter rank and expert adapter rank) to 64. We found that assigning rank 48 to the shared adapter and rank 16 to each expert adapter achieves a good trade-off between learning common and specific programming language knowledge. To study whether a larger or smaller shared adapter is beneficial, we experiment with the following distributions of shared adapter and expert adapters:

**MoLE-64+0** uses shared adapter with rank 64 and no expert adapter. This variant learns only common programming language knowledge (but separeted from natural language knowledge which is handled by the NL adapter).

**MoLE-56+8** uses shared adapter with rank 56 and expert adapters with rank 8.

**MoLE-48+16** (the default MoLE model) uses shared adapter with rank 48 and expert adapters with rank 16.

**MoLE-32+32** uses shared adapter with rank 32 and expert adapters with rank 32, to encourage the learning of language-specific knowledge.

### C. Metrics

Following HumanEvalPack [2], [28], we use Pass@1 consistently as the metric on summarization, synthesis, and translation tasks. We use greedy decoding to generate one solution per problem. For synthesis and translation tasks, the generated code solution is executed against the test cases that comes

TABLE II: Summary of Pass@1 results on all three tasks averaged over all programming languages. The best parameter-efficient finetuning model (excluding Full-FT) is highlighted.

| Model | Summarization | Synthesis | Translation |
|---|---|---|---|
| No-FT | 16.67 | 29.07 | 48.84 |
| AllLang-FT | 22.43 | 30.62 | 59.78 |
| PerLang-FT | 22.60 | 30.96 | N/A |
| Full-FT | 23.71 | 33.98 | 60.64 |
| MoLE | **23.75** | **31.34** | **60.62** |

with the HumanEvalPack benchmark to verify its correctness; Pass@1 is the percentage of problems that the generated solution passes all test cases. For summarization task, the generated natural language summary is fed into the model again to synthesize a code snippet, whose correctness is verified by the test cases; Pass@1 is the percentage of problems where the code snippet synthesized from the generated natural language summary passes all test cases.

### *D. Experiment Setup*

**Finetuning**. We finetune all models and baselines (except for No-FT) for two epochs using the AdamW optimizer. We set the batch size to 64, and use a linear learning rate scheduler with 0.05 warmup. The learning rate is set to 1e-4, which is based on a parameter search when finetuning Full-FT and AllLang-FT with learning rates of {2e-5, 5e-5, 1e-4, 2e-4}. Each finetuning is repeated three times using different initial random seeds to account for the randomness of the training process, and we report the average results.

**Hardware and Software Environment**. We conduct finetuning and evaluation on a GPU cluster with two types of GPUs: Nvidia A100 (40GB) and Nvidia Ada6000 (48GB); each finetuning or evaluation job is executed on one GPU. We use Python 3.9, PyTorch 2.1.2, and Transformers 4.42.4.

### *E. Results*

Table II presents the summary of results on all three tasks of code summarization, synthesis, and translation, where the numbers are the average Pass@1 over all data samples across all programming languages on the corresponding task.

We found that MoLE outperforms both AllLang-FT and PerLang-FT on all three tasks, confirming MoLE's ability in assisting multilingual programming. As expected, finetuning one model per programming language (i.e., PerLang-FT) outperforms sharing a single model for all programming languages (i.e., AllLang-FT), but is only marginally better. Compared to PerLang-FT which finetunes each LoRA adapter on a smaller subset, the expert adapters in MoLE are jointly trained, and thus can achieve a better organization of specific programming language knowledge, and benefit from the shared adapter and NL adapter for processing common programming language and natural language knowledge. Notably, MoLE's performance is on a par with the full finetuning baseline (Full-FT), even outperforming it on the summarization task, which shows the effectiveness of MoLE's architecture under the parameter-efficient finetuning paradigm.

TABLE III: Pass@1 results on the summarization task for each programming language. The best parameter-efficient finetuning model (excluding Full-FT) is highlighted.

| Model | C++ | Go | Java | JS | Python | Rust | Avg |
|---|---|---|---|---|---|---|---|
| No-FT | 16.46 | 8.54 | 29.88 | 15.24 | 19.51 | 10.37 | 16.67 |
| AllLang-FT | **23.17** | 19.11 | 29.27 | 20.12 | 31.50 | 11.38 | 22.43 |
| PerLang-FT | 20.53 | 16.46 | 30.69 | 22.76 | **33.54** | 11.59 | 22.60 |
| Full-FT | 22.76 | 20.33 | 27.03 | 26.42 | 31.50 | 14.23 | 23.71 |
| MoLE-64+0 | 22.36 | 18.09 | 27.64 | 22.97 | 31.50 | 12.40 | 22.49 |
| MoLE-56+8 | 22.97 | 20.73 | 29.07 | 22.56 | 31.91 | 11.99 | 23.21 |
| MoLE-48+16 | 22.76 | 19.31 | **31.30** | 21.75 | **33.54** | **13.82** | **23.75** |
| MoLE-32+32 | 20.93 | **21.14** | 30.69 | **24.19** | 31.50 | 13.01 | 23.58 |

TABLE IV: Pass@1 results on the synthesis task for each programming language. The best parameter-efficient finetuning model (excluding Full-FT) is highlighted.

| Model | C++ | Go | Java | JS | Python | Rust | Avg |
|---|---|---|---|---|---|---|---|
| No-FT | **33.54** | 26.22 | 31.71 | 32.32 | 33.54 | 17.07 | 29.07 |
| AllLang-FT | 30.89 | 27.03 | 36.18 | 36.59 | 35.37 | 17.68 | 30.62 |
| PerLang-FT | 32.93 | 24.80 | **38.82** | 33.13 | 34.76 | **21.34** | 30.96 |
| Full-FT | 35.37 | 28.86 | 37.80 | 39.43 | 39.84 | 22.56 | 33.98 |
| MoLE-64+0 | 31.50 | 25.41 | 35.98 | 34.15 | 34.96 | 18.29 | 30.05 |
| MoLE-56+8 | 32.11 | **27.24** | 35.16 | 34.96 | 35.77 | 20.73 | 31.00 |
| MoLE-48+16 | 31.30 | **27.24** | 36.79 | 36.59 | **36.18** | 19.92 | 31.34 |
| MoLE-32+32 | 30.49 | 26.63 | 38.21 | **38.21** | 35.98 | 20.33 | **31.64** |

Tables III and IV present the results per programming language for the summarization and synthesis tasks, respectively. We found that MoLE outperforms the baselines in most cases, especially on low-resource languages such as Go and JS. For example, on the summarization task, Pass@1 of JS is improved from 22.76% (of PerLang-FT) up to 24.19% (of MoLE-32+32); on the synthesis task, Pass@1 of JS is improved from 33.13% (of PerLang-FT) up to 38.21% (of MoLE-32+32). For Python, the performance of MoLE is the same as or slightly better than that of baselines, which can be attributed to the fact that Python is prevalent in the pretraining dataset of the base model. We observe minor performance drops for C++, Java on synthesis task, and Rust on summarization task when using MoLE; we hypothesize that these "harder" programming languages require stronger language-specific knowledge to excel, and our finetuning data may be insufficient to train good expert adapters for them. Nevertheless, MoLE still achieves an overall improvement over the baselines when considering the average of all programming languages, making it a suitable choice for multilingual programming.

Fig. 5 presents the results on the translation task for each pair of source and target programming language. We observe a similar trend as in the summarization and synthesis tasks, where MoLE achieves the most improvements over the baselines on low-resource languages. For example, when translating from other programming languages to Rust, MoLE achieves an average Pass@1 of 37.76%, outperforming the AllLang-FT baseline's 36.87%; when translating from Rust to other programming languages, the improvement is even larger, with MoLE's 62.52% vs. AllLang-FT's 59.76%.

**Answer to RQ1**. MoLE outperforms both AllLang-FT and

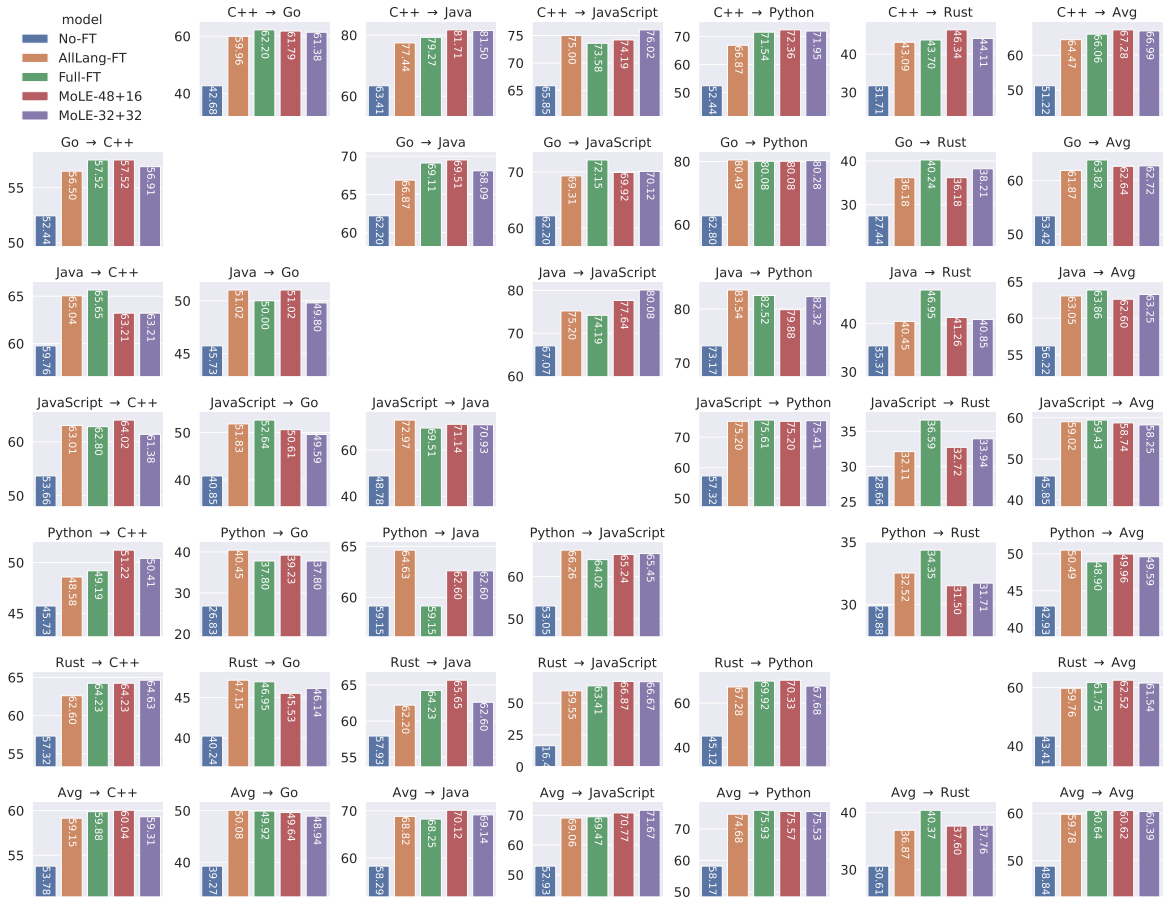


Fig. 5: Barplots showing Pass@1 on the translation task for each source-target programming language pair. The last column shows the average Pass@1 when translating *from* a programming language to others; the last row shows the average Pass@1 when translating from others *to* a programming language.

PerLang-FT baselines on code summarization, synthesis, and translation tasks. This indicates that MoLE's architecture allows more efficient finetuning of multilingual programming models. In particular, MoLE can greatly benefit low-resource languages by jointly training with other popular languages.

Next, we study the impact of the rank distribution between the shared adapter and expert adapters on the performance of MoLE. The lower parts of Tables III and IV present the Pass@1 of MoLE variants with different rank distributions. We found that not using expert adapters at all (i.e., MoLE-64+0) leads to worse performance, indicating that learning specific programming language knowledge is important in MoLE's architecture. The other three variants are have comparable performance; MoLE-48+16 is the best on the summarization task, and MoLE-32+32 is more effective on the synthesis task. On the translation task, we compare the performance of MoLE-48+16 and MoLE-32+32 variants, as shown by the last two bars in the barplots in Figure 5. MoLE-48+16 outperforms MoLE-32+32 by a small margin, with same or better performance on 18 out of 30 source-target language pairs.

**Answer to RQ2**. The use of expert adapters is critical in improving MoLE's multilingual programming ability. Overall, the MoLE-48+16 variant achieves the best performance on two out of the three tasks (summarization and translation) and is comparable to the best performing variant on the synthesis task (MoLE-32+32), thus we recommend MoLE-48+16 as the default MoLE configuration.

We perform an ablation study to investigate the impact of different design choices in MoLE, the results of which are presented in Table V. MoLE adapted the PiSSA [27] initialization technique for the LoRA adapters, which promotes a better organization of language-agnostic and language-specific knowledge and speeds up the convergence of the finetuning process. Compared with the standard LoRA initialization (StdInit), the final performance is close, favoring the MoLE on the summarization task and the Go and Python programming languages. MoLE also positions the shared adapter before the expert adapters in the principle components' space, with the hypothesis that the language-agnostic knowledge is more important than the language-specific knowledge. The Shared-Last variant explores the reversed order by positioning the shared adapter after the expert adapters. Results show that MoLE consistently outperforms SharedLast on both tasks,

TABLE V: Ablation study on the design choices of MoLE: StdInit uses the standard LoRA initialization instead of PiSSA; SharedLast assigns the most significant components to the expert adapters instead of the shared adapter; NLExpert processes natural language using an expert adapter.

(a) Pass@1 on the summarization task.

| Model | C++ | Go | Java | JS | Python | Rust | Avg |
|---|---|---|---|---|---|---|---|
| MoLE | 22.76 | 19.31 | 31.30 | 21.75 | 33.54 | 13.82 | 23.75 |
| StdInit | 23.37 | 16.26 | 28.86 | 23.58 | 32.72 | 14.84 | 23.27 |
| SharedLast | 21.14 | 16.46 | 24.59 | 18.09 | 27.64 | 14.23 | 20.36 |
| NLExpert | 21.34 | 19.11 | 30.08 | 22.15 | 30.49 | 13.21 | 22.73 |

(b) Pass@1 on the synthesis task.

| Model | C++ | Go | Java | JS | Python | Rust | Avg |
|---|---|---|---|---|---|---|---|
| MoLE | 31.30 | 27.24 | 36.79 | 36.59 | 36.18 | 19.92 | 31.34 |
| StdInit | 33.33 | 25.81 | 40.24 | 37.40 | 34.55 | 19.51 | 31.81 |
| SharedLast | 31.50 | 25.81 | 37.20 | 35.98 | 34.55 | 18.50 | 30.59 |
| NLExpert | 31.71 | 26.42 | 36.99 | 33.94 | 34.76 | 19.31 | 30.52 |

confirming our hypothesis. Finally, MoLE treats natural language specially by using a separate rank-64 NL adapter, since the natural language knowledge should be separated from programming language knowledge. To verify this intuition, we examine NLExpert, which processes natural language with an additional expert adapter instead of a separate NL adapter (i.e. treat natural language as one of the programming languages). Overall, MoLE outperforms NLExpert, with larger gap on the summarization task than the synthesis task, indicating that it is more beneficial to learn natural language knowledge in a separate adapter.

**Answer to RQ3**. In MoLE's architecture, the order of the shared adapter and expert adapters in the principle components' space is critical, and a separate NL adapter is essential for processing natural language knowledge.

We further investigate whether each expert adapter in MoLE is properly learning its assigned programming language. To quantitatively measure this, we manually activate a misassigned adapter during inference (using the same finetuned MoLE model), and compare the performance with when the correct adapter is activated. The results are shown in Figure 6. On average (last column in Figure 6), activating the incorrect expert adapter leads to worse performance in most of the cases, with the exception of the expert adapter for Go and Python. We suspect that these two expert adapters are stronger than others for different reasons: Go is syntactically close to several other programming languages being studied (e.g., C++, Java, and Rust), thus using it to summarize or generate code for other programming languages achieves moderate performance; the Python expert adapter is the most well-trained one, given that Python data takes the largest proportion in both the finetuning dataset of MoLE and the pretraining dataset of the base model. Looking at the detailed results on each programming language, we found that applying the expert adapter of one programming language to a similar programming language's input/output will likely achieve good performance, e.g., using C++ expert adapter on Go and verse versa both achieve similar or higher performance than using the correct expert adapter.

We also experimented with using only the NL adapter throughout the inference, i.e., considering all tokens as natural language, as shown by the "NL" bar in Figure 6. Except for a few outliers (e.g., Java on the synthesis task), the performance of forcing the NL adapter is lower than using the correct adapter, indicating that the natural language knowledge and programming language knowledge are also properly disentangled in MoLE.

**Answer to RQ4**. Our analysis on the Pass@1 when activating MoLE's expert adapters on mismatching languages confirms that each expert adapter is learning language-specific knowledge as expected. In some cases, the expert adapter for syntactically similar programming languages (e.g., C++ and Go) can be exchanged without significant performance drop.

## V. Limitations and Future Work

MoLE is the first architecture designed for multilingual programming with explicit expert adapter modules. Constrained by the accessible computational resources, our implementation is based on an 1.3B LLM and uses LoRA for parameter-efficient finetuning. The same idea can be experimented on larger LLMs, extended to full finetuning, or even applied in the pretraining phase. We believe that the proposed shared-and-expert adapters architecture will lead to better multilingual programming performance on larger-scale LLMs.

The current finetuning dataset of MoLE is limited in its scale and modality. Our results analysis shows that some of MoLE's expert adapters are not sufficiently trained compared to the programming languages with sufficient data (e.g., Python). Combining similar languages into a single expert adapter can further improve parameter efficiency while alleviating the under-training of low-resource languages. Most samples in the finetuning dataset also involve only one programming language. Having multiple programming languages in one data sample allows backpropagation of gradients across different expert adapters, which can further improve knowledge sharing and specialization. Code translation may be one source of such multilingual data, however, our preliminary experiments show that the quality of finetuning data is critical. In the future, we will identify high-quality multilingual finetuning datasets to improve MoLE's performance.

The ideal usage scenario of MoLE is as an IDE/editor extension, where the extension is initialized with one copy of base model, shared adapter, and NL adapter, and the user can download only the expert adapters for the programming languages that they plan to use (which are much smaller than downloading a language-specific finetuned model without using MoLE). We plan to explore MoLE's ability to finetune newly added expert adapters in future work.

## VI. Related Work

**LLMs for Multilingual Programming**. Most LLMs for code in recent years [1], [5], [9], [25], [35], [42], including the

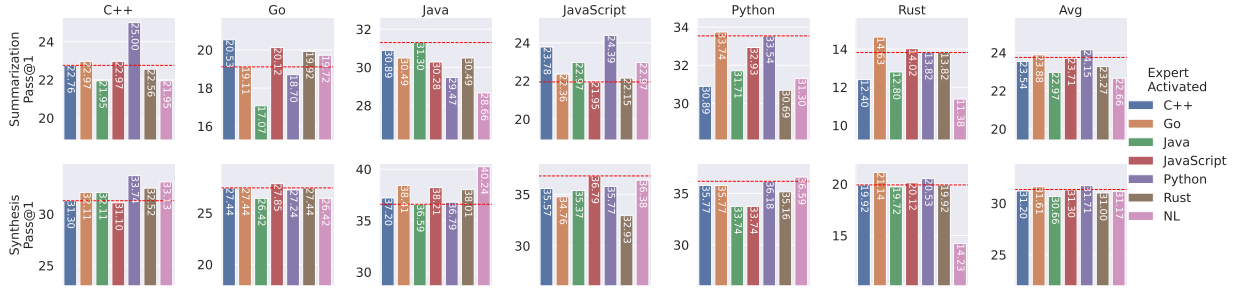


Fig. 6: Performance of MoLE when forcing the model to activate one expert adapter or NL adapter in the summarization (top) and synthesis (bottom) tasks. The red dotted line in each subplot represents the performance of the MoLE variant when activating the correct adapter matching the input/output.

base model we used in our study—DeepSeek Coder [15], are pretrained on code from multiple programming languages, thus are multilingual by natural. Researchers have been exploring applying LLMs to code translation [23], [26], [39], a task involving two programming languages. Xu et al. [36]'s PolyCoder was among the first to systematically evaluate the performance of multilingual LLMs for code. However, all existing work uses the default transformer architecture, and ours is the first to propose an architecture that optimizes the efficiency of training multilingual LLMs for code.

To evaluate code LLMs' performance across multiple programming languages, a number of multilingual benchmarks have been created, such as HumanEvalPack [28] (which we used in our study), MultiPL-E [8], and HumanEval-X [42]. They all come with a diverse set of programming languages and executable test cases to evaluate the correctness of model outputs (following the practice of their origin dataset, HumanEval [9]). We adopted HumanEvalPack in our study due to its multiple programming tasks and having consistent number of samples across all programming languages.

**Parameter-Efficient Finetuning of LLMs**. Low-rank adaptation (LoRA) and their variants [10], [17], [20], [21], [24], [32], [40] are widely used for the efficient finetuning of LLMs by constraining trainable parameters in a low rank space. In this work, we use LoRA adapters to hold language-agnostic and language-specific trainable parameters. Researchers have explored improving the initialization strategy of LoRA by using principle component analysis [7], [27], which we also adopted in MoLE to bootstrap the distribution of programming-language-related knowledge among the adapters.

**Mixture-of-Experts (MoE) Architecture**. The MoE architecture scales up the number of parameters by using a set of expert modules that will be conditionally activated based on input tokens, and has been shown to be effective on many tasks including programming [12]–[14], [19], [29], [30], [37], [43]–[45]. Recent work also proposed using LoRA in MoE [11], [18], [38], [41], which motivated the design of MoLE. Our MoLE architecture is in spirit similar to MoE, but with a few key differences: (1) the expert modules in MoE are not pre-assigned to specific tasks and all experts must be present during inference, while each expert adapter in MoLE is in charge of a specific programming language and only the relevant experts are needed during inference; (2) MoE learns a "router" gating network to determine which expert to activate, while MoLE determines which expert adapter to activate based on prior context or user specification (Section II-C). MoE architecture can also be initialized from an existing pretrained LLM, using a technique known as upcycling [16], [19]; a common limitation of upcycling is the resulting expert modules become very similar, which inspired the design of shared adapter in MoLE.

## VII. Conclusion

We proposed MoLE, the first model architecture designed for finetuning code LLMs for multiple programming languages. By using a combination of a shared adapter, expert adapters, and an NL adapter, MoLE balances efficiency and specialization for multilingual programming. The adapters are initialized according to a principal-components-based strategy, leveraging the pretrained model's knowledge, and are jointly finetuned on a dataset spanning multiple programming languages. At inference, the appropriate expert adapter is dynamically activated based on the token being processed. We performed an extensive evaluation on a benchmark with three tasks: code summarization, synthesis, and translation. We demonstrated that MoLE is more effective than the baselines of finetuning a single model shared by all programming languages, or finetuning multiple language-specific models, and is even close to that of a fully finetuned model. We also confirmed the effectiveness of our proposed architecture in disseminating knowledge across multiple programming languages. We envision that MoLE will be more powerful when applied on larger models trained with more computational resources, which we will explore in the future.

## Acknowledgments

We thank Yu Liu and the anonymous reviewers for their comments and feedback. This work is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) under funding reference number RGPIN-2024-04909 and the University of Waterloo start-up grant.

## REFERENCES


[1] "Amazon CodeWhisperer," https://platform.qa.com/course/amazon-codewhisperer-generating-code-ai-4679.

[2] "bigcode/humanevalpack," https://huggingface.co/datasets/bigcode/humanevalpack.

[3] "Cursor - the AI code editor," https://www.cursor.com/.

[4] "deepseek-ai/deepseek-coder-1.3b-base," https://huggingface.co/deepseek-ai/deepseek-coder-1.3b-base.

[5] "GitHub Copilot," https://github.com/features/copilot.

[6] "glaiveai/glaive-code-assistant-v3," https://huggingface.co/datasets/glaiveai/glaive-code-assistant-v3.

[7] K. Bałazy, M. Banaei, K. Aberer, and J. Tabor, "LoRA-XS: Low-rank adaptation with extremely small number of parameters," 2024.

[8] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, A. Guha, M. Greenberg, and A. Jangda, "MultiPL-E: A scalable and polyglot approach to benchmarking neural code generation," *TSE*, vol. 49, no. 7, pp. 3675–3691, 2023.

[9] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[10] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient finetuning of quantized llms," in *NeurIPS*, 2023, pp. 10 088–10 115.

[11] S. Dou, E. Zhou, Y. Liu, S. Gao, W. Shen, L. Xiong, Y. Zhou, X. Wang, Z. Xi, X. Fan, S. Pu, J. Zhu, R. Zheng, T. Gui, Q. Zhang, and X. Huang, "LoRAMoE: Alleviating world knowledge forgetting in large language models via MoE-style plugin," in *ACL*, 2024, pp. 1932–1945.

[12] N. Du, Y. Huang, A. M. Dai, S. Tong, D. Lepikhin, Y. Xu, M. Krikun, Y. Zhou, A. W. Yu, O. Firat, B. Zoph, L. Fedus, M. Bosma, Z. Zhou, T. Wang, Y. E. Wang, K. Webster, M. Pellat, K. Robinson, K. Meier-Hellstern, T. Duke, L. Dixon, K. Zhang, Q. V. Le, Y. Wu, Z. Chen, and C. Cui, "GLaM: Efficient scaling of language models with mixture-of-experts," in *ICML*, 2022, pp. 5547–5569.

[13] W. Fedus, J. Dean, and B. Zoph, "A review of sparse expert models in deep learning," 2022.

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

[15] 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," 2024.

[16] E. He, A. Khattar, R. Prenger, V. Korthikanti, Z. Yan, T. Liu, S. Fan, A. Aithal, M. Shoeybi, and B. Catanzaro, "Upcycling large language models into mixture of experts," 2024.

[17] E. J. Hu, yelong shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-rank adaptation of large language models," in *ICLR*, 2022.

[18] C. Huang, Q. Liu, B. Y. Lin, T. Pang, C. Du, and M. Lin, "LoraHub: Efficient cross-task generalization via dynamic LoRA composition," in *COLM*, 2024.

[19] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. d. l. Casas, E. B. Hanna, F. Bressand *et al.*, "Mixtral of experts," 2024.

[20] D. Kalajdzievski, "A rank stabilization scaling factor for fine-tuning with LoRA," 2023.

[21] D. J. Kopiczko, T. Blankevoort, and Y. M. Asano, "VeRA: Vector-based random matrix adaptation," in *ICLR*, 2024.

[22] V. Kulkarni and S. Reddy, "Separation of concerns in model-driven development," *IEEE Software*, vol. 20, no. 5, pp. 64–69, 2003.

[23] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," in *NeurIPS*, 2020.

[24] 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 *ICML*, 2024, pp. 32 100–32 121.

[25] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "WizardCoder: Empowering code large language models with evol-instruct," in *ICLR*, 2024.

[26] M. Macedo, Y. Tian, P. Nie, F. R. Cogo, and B. Adams, "InterTrans: Leveraging transitive intermediate translations to enhance LLM-based code translation," in *ICSE*, 2025, p. to appear.

[27] F. Meng, Z. Wang, and M. Zhang, "PiSSA: Principal singular values and singular vectors adaptation of large language models," in *NeurIPS*, 2024, pp. 121 038–121 072.

[28] N. Muennighoff, Q. Liu, A. R. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. V. Werra, and S. Longpre, "OctoPack: Instruction tuning code large language models," in *ICLR*, 2024.

[29] N. Muennighoff, L. Soldaini, D. Groeneveld, K. Lo, J. Morrison, S. Min, W. Shi, P. Walsh, O. Tafjord, N. Lambert, Y. Gu, S. Arora, A. Bhagia, D. Schwenk, D. Wadden, A. Wettig, B. Hui, T. Dettmers, D. Kiela, A. Farhadi, N. A. Smith, P. W. Koh, A. Singh, and H. Hajishirzi, "OLMoE: Open mixture-of-experts language models," 2024.

[30] J. Puigcerver, C. Riquelme, B. Mustafa, and N. Houlsby, "From sparse to soft mixtures of experts," in *ICLR*, 2024.

[31] B. Shen, J. Zhang, T. Chen, D. Zan, B. Geng, A. Fu, M. Zeng, A. Yu, J. Ji, J. Zhao *et al.*, "PanGu-Coder2: Boosting large language models for code with ranking feedback," 2023.

[32] C. Si, X. Wang, X. Yang, Z. Xu, Q. Li, J. Dai, Y. Qiao, X. Yang, and W. Shen, "FLoRA: Low-rank core space for N-dimension," 2024.

[33] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *NeurIPS*, vol. 30, 2017.

[34] 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 *ICSE*, 2023, pp. 5–16.

[35] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Empowering code generation with OSS-instruct," in *ICML*, 2024.

[36] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *MAPS*, 2022, pp. 1–10.

[37] F. Xue, Z. Zheng, Y. Fu, J. Ni, Z. Zheng, W. Zhou, and Y. You, "OpenMoE: An early effort on open mixture-of-experts language models," in *ICML*, 2024.

[38] T. Zadouri, A. Üstün, A. Ahmadian, B. Ermis, A. Locatelli, and S. Hooker, "Pushing mixture of experts to the limit: Extremely parameter efficient moe for instruction tuning," in *ICLR*, 2024.

[39] J. Zhang, P. Nie, J. J. Li, and M. Gligoric, "Multilingual code co-evolution using large language models," in *FSE*, 2023, pp. 695–707.

[40] L. Zhang, L. Zhang, S. Shi, X. Chu, and B. Li, "LoRA-FA: Memory-efficient low-rank adaptation for large language models fine-tuning," 2023.

[41] L. Zhao, W. Zeng, S. Xiaofeng, and H. Zhou, "MoSLD: An extremely parameter-efficient mixture-of-shared LoRAs for multi-task learning," in *COLING*, 2025, pp. 1647–1659.

[42] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li, T. Su, Z. Yang, and J. Tang, "CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X," in *KDD*, 2023, pp. 5673–5684.

[43] Y. Zhou, N. Du, Y. Huang, D. Peng, C. Lan, D. Huang, S. Shakeri, D. So, A. M. Dai, Y. Lu, Z. Chen, Q. V. Le, C. Cui, J. Laudon, and J. Dean, "Brainformers: Trading simplicity for efficiency," in *ICML*, 2023, pp. 42 531–42 542.

[44] Y. Zhou, T. Lei, H. Liu, N. Du, Y. Huang, V. Zhao, A. Dai, Z. Chen, Q. Le, and J. Laudon, "Mixture-of-experts with expert choice routing," in *NeurIPS*, 2022, pp. 7103–7114.

[45] B. Zoph, I. Bello, S. Kumar, N. Du, Y. Huang, J. Dean, N. Shazeer, and W. Fedus, "ST-MoE: Designing stable and transferable sparse expert models," 2022.