MultiPL-MoE: Multi-Programming-Lingual Extension of Large Language Models through Hybrid Mixture-of-Experts

Qing Wang, Xue Han*, Jiahui Wang, Lehao Xing
Qian Hu, Lianlian Zhang, Chao Deng, Junlan Feng*
JIUTIAN Team, China Mobile Research Institute, Beijing, China
{wangqingai,hanxueai, wangjiahui, xinglehao, huqianai, zhanglianlian, dengchao, fengjunlan}@chinamobile.com

Abstract

Despite LLMs' excellent code creation capabilities, multilingual code generation remains extremely challenging. To address this, we intent to improve the multi-programming-lingual (MultiPL) performance of the base LLMs while retaining the most popular ones using restricted computational resources. We consider MultiPL to be a special case of multiple natural languages and propose a MultiPL extension of LLMs utilizing a hybrid mixture of experts (MoE), called MultiPL-MoE. Specifically, MultiPL-MoE combines two paired MoEs to optimize expert selection at both the token and segment levels. The **token-level MoE** is a standard upcycling MoE structure with a shared expert and a novel gate weight normalization approach that aids in the final fusion with the segment-level MoE. The **segment-level MoE** incorporates two innovative designs to better capture the syntactic structure and contextual patterns of programming languages: First, using a sliding window to partition the input token sequence into multiple segments; Then, adopting an expert-choice routing strategy that allows experts to select the top-k segments. The results of the experiment proved the effectiveness of MultiPL-MoE[1].

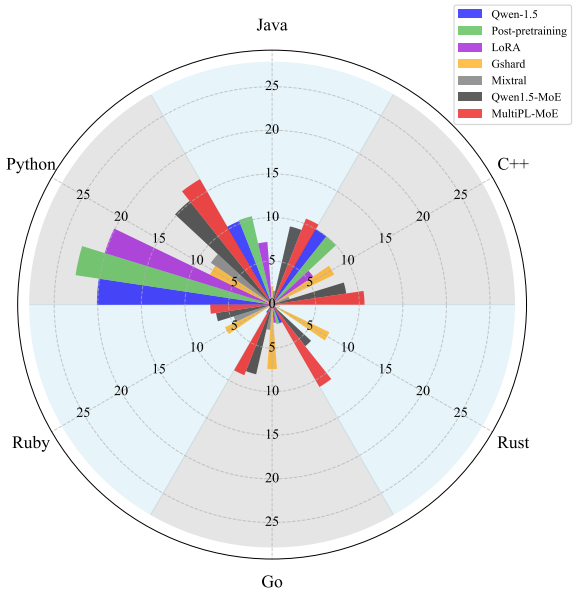

Figure 1: MultiPL-MoE achieves dual success on HumanEval by preserving proficiency in high-resource programming languages and advancing performance on low-resource languages. Specially, it attains a pass@1 score surpassing the second-best model by 1.97 points on low-resource languages while sacrificing only 0.9 points below Qwen1.5 on high-resource languages.

1 Introduction

Large Language Models (LLMs) such as GPT4 (Achiam et al., 2023) and Qwen2.5-coder (Hui et al., 2024) have shown remarkable capabilities in code generation, facilitating software engineers to boost their efficiency. However, many studies have highlighted a significant discrepancy between performance on high-resource programming languages such as Python and low-resource ones like Rust (Joel et al., 2024), which refers to those with limited availability both online and in training datasets.

The growing demand for multilingual code generation capabilities has driven efforts to inject extensive programming language knowledge into LLMs. To enhance the base LLMs' multi-programming-lingual (**MultiPL**) capabilities, some existing methods (Xu et al., 2022; Fu et al., 2024; Para et al., 2021; Pujar et al., 2023; Han et al., 2023; Liguori et al., 2024; Han et al., 2025) continue train a base LLM with data from multiple programming languages but are highly computationally intensive. Others (Muennighoff et al., 2023; Cui et al., 2024; Fakih et al., 2024; Wen and Chaudhuri, 2023; Yang et al., 2023; Zhang et al., 2024) improve specific programming languages by fine-tuning a base LLM with specific high-quality low-resource data. How-

*The corresponding author.
[1] https://github.com/Eduwad/MultiPL-MoE

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 12817–12828
November 4-9, 2025

ever, they suffer from severe catastrophic forgetting for the base model's existing code capabilities.

Facing these challenges, we intend to improve LLMs' MultiPL performance, especially for low-resource programming languages, while maintaining the performance of existing popular ones with a relatively small additional computational budget. Recent advances in programming language translation have demonstrated that they share common semantics (Khan et al., 2024) and can be converted from one to another in the same way that natural languages do (Zhu et al., 2024a). Consequently, we may enhance the MultiPL performance of LLMs by leveraging recent advancements in natural language expansion technology, which adopt Mixture-of-Experts (MoE) architecture by upcycling methods and prove to achieve good language expansion effectively while preventing catastrophic forgetting (Zhou et al., 2024).

In this paper, we take MultiPL as a special case of multiple natural language expansion and propose MultiPL-MoE, a multi-programming-lingual (MultiPL) extension of LLMs through hybrid mixture-of-experts (MoE). Unlike prior MoE methods for multiple natural language extension (Zhou et al., 2024) that focus on internal language consistency by predicting subsequent tokens but ignore the internal syntax structures of programming languages (Tehrani et al., 2024), MultiPL-MoE employs novel hybrid MoE architecture for fine-grained learning of both the token-level semantic features and segment-level syntax features, enabling the model to reason about code structure by identifying and categorizing different syntactic elements.

The token-level MoE combines traditional token-choice routing with a shared expert (Dai et al., 2024; Ding et al., 2024) and a novel routing weight normalization mechanism to handle scale mismatch during the later fusion with the segment-level MoE.

In particular, MultiPL-MoE combines two paired MoEs to optimize expert selection at both the token and segment levels. The token-level MoE is a typical upcycling MoE structure with a shared expert (Dai et al., 2024; Ding et al., 2024; Gou et al., 2023) and a novel gate weight normalization approach that aids in the final fusion with the segment-level MoE. To better capture the syntactic structure and contextual patterns of programming languages, the segment-level MoE incorporates two innovative designs: 1) Using a sliding window to partition the input token sequence into multiple segments. 2) Adopting a different routing strategy with segment-level MoE that allows experts to select the top-k segments instead of segments selecting experts.

As illustrated in Figure 1, experiment results show that MultiPL-MoE achieves balanced MultiPL proficiency with 1.97% higher average accuracy than baselines on low-resource languages (Rust, Go, Ruby) in the HumanEval benchmark, while matching state-of-the-art performance within a 1.80% gap for high-resource languages like Python, Java, and C++.

2 Related Work

Multilingual code LLM. Recent work has developed specialized language models for technical domains, including PolyCoder (Xu et al., 2022) (multi-language code generation), Hardware Phi-1.5B (Fu et al., 2024) (hardware domain), SketchGen (Para et al., 2021) (CAD sketch), Ansible Wisdom (Pujar et al., 2023) (YAML automation), and adversarial PowerShell code generation (Liguori et al., 2024), which are highly computationally intensive for domain-specific adaptations. Other studies have addressed programming language challenges through diverse approaches, including multilingual model finetuning (Muennighoff et al., 2023), code-to-code augmentation for data-scarce languages (Cui et al., 2024)), and verifiable PLC code synthesis in industrial contexts (Fakih et al., 2024).

Mixture-of-Experts (MoE). Scaling large language models (LLMs) is critical to enhance linguistic capabilities. MoE can achieve scalable capacity expansion with sub-linear computation costs (Shazeer et al., 2017), which replaces dense FFN layers with multiple experts and a router. GShard (Lepikhin et al., 2020), Switch Transformer (Fedus et al., 2022), ST-MoE (Zoph et al., 2022), and GLaM (Du et al., 2022) require full training from scratch. Sparse upcycling (Komatsuzaki et al., 2022) offers a cost-effective alternative by converting pretrained dense models into sparsely activated MoE models (e.g., Skywork-MoE (Wei et al., 2024), LlaMA-MoE (Zhu et al., 2024b)). Several works mainly focused on establishing adaptive routing mechanisms to optimize expert allocation. Shazeer et al. (2017) introduces top-k experts routing with auxiliary loss. Expert choice routing (Zhou et al., 2022) independently selects the top-k tokens with regularization mechanism. However, the above work simultaneously redundant knowledge integration. Recent work (Dai et al., 2024;

Xue et al.; Team, 2024b; Gou et al., 2023) proposes always-activated shared experts to address this limitation.

3 Methodology

Figure 2 illustrates the overall framework of MultiPL-MoE, which is constructed through upcycling the original FFN layer of a base LLM. The token-level MoE is a conventional token-choice routing combined with a shared expert (Dai et al., 2024; Ding et al., 2024) with a novel routing weight normalization method to address scale mismatch during the later fusion with the segment-level MoE. For the segment-level MoE, we adopt the expert-choice routing mechanism with the input as contextually coherent segments, enabling experts to capture the syntax structures and discourse-level features. The outputs of the two MoEs are finally fused together.

3.1 Token-level MoE

Similar to (Dai et al., 2024), the token-level MoE includes a shared expert in each layer to capture common knowledge and reduce redundancy in routed experts. The parameters of the shared expert are fixed during training. The routing strategy is traditional token-choice routing, in which the token selects experts.

Compared with the standard Transformer, token-level MoE replaces each Feed-Forward Network (FFN) layer with a MoE layer, which uses N^{tok} expert networks that are structurally equivalent to the original FFN layer. A router directs each input token to K out of N^{tok} expert networks. $g_{i,t}$ refers to the gate value for the i-th expert given the t-th token. Formally, for the l-th MoE layer, output hidden state $O^{tok}_{t,l}$ of the t-th input token is computed as follows (Dai et al., 2024).

$$
\begin{aligned}
O^{tok}_{t,l} &= \sum_{i=1}^{N^{tok}} (g_{i,t} FFN_i(x_t)) \\
g_{i,t} &= \begin{cases} s_{i,t}, & s_{i,t} \in TopK(S_t, K) \\ 0, & otherwise \end{cases} \\
S_t &= \{s_{i,t} | 1 \le i \le N^{tok}\} \\
s_{i,t} &= Softmax_i(x_t \cdot W_{t\_expert})
\end{aligned}
\tag{1}
$$

where x_t refers to the hidden states of the t-th input token for the l-th MoE layer. $TopK(\cdot)$ refers to a function extracting K highest scores. $s_{i,t}$ refers to the affinity score between the i-th expert and the t-th token. W_{t_expert} denotes the router matrix.

However, the scale mismatch between the upcycled MoE layer and the original FFN layer can lead to performance degradation (Wu et al., 2022). This issue could be even worse for our hybrid MoE architecture because we must finally fuse the outputs of token-level and segment-level MoEs. To facilitate the final fusion, inspired by (Feldhus, 2024), we enforced the restriction that the sum of gate value $g_{i,t}$ for the selected top-K experts equals 1. Specifically, we first employ the router network to calculate the affinity score of the shared expert for token t, denoted as $s_{1,t}$, and select the highest $K-1$ affinity score from the remaining scores. The sum of K affinity scores is then used as a scaling factor $Norm$ to normalize them, ensuring $\sum g_{i,t} = 1$. Therefore, the gate value of $g_{i,t}$ can be updated mathematically as follows:

$$
\begin{aligned}
g_{i,t} &= \begin{cases} \frac{1}{Norm} s_{1,t}, & s_{1,t} \in S'_t \\ \frac{1}{Norm} s_{i,t}, & s_{i,t} \in S_{K,t} \\ 0, & otherwise \end{cases} \\
S_{K,t} &= TopK(S'_t, K-1) \\
S'_t &= \{s_{i,t} | 1 < i \le N^{tok}\} \\
s_{i,t} &= Softmax_i(x_t \cdot W_{t\_expert}) \\
Norm &= \sum S_{K,t} + s_{1,t}
\end{aligned}
\tag{2}
$$

where $S_{K,t}$ denotes the collection of $K-1$ highest router scores among all the experts excluding the shared expert.

3.2 Segment-level MoE

Given that experts tend to be underspecialized with token-choice routing (Zhou et al., 2022), we hypothesize that such pitfalls may be more severe in learning the segment structures of the programming languages. Therefore, the segment-level MoE adopts the expert-choice routing strategy that independently selects top-K segments for each expert.

Given a total number of T tokens in an input sample $S = \{token_1, token_2, \ldots, token_T\}$, we first partitioned S into P segments based on the context window a. These segments are denoted as $SEG = \{seg_1, \ldots, seg_p, \ldots, seg_P\}$, where $seg_p \in \mathbb{R}^{a \times hidden\_size}$, $P = [\frac{T}{a}]$. The expert capacity r, which represents the number of segments each expert can take, is defined as follows:

$$
r = \frac{V \times c}{N^{seg}}, V = \mathcal{B} \times P \tag{3}
$$

where V is the total number of segments within an input batch. Notice that a batch of input includes

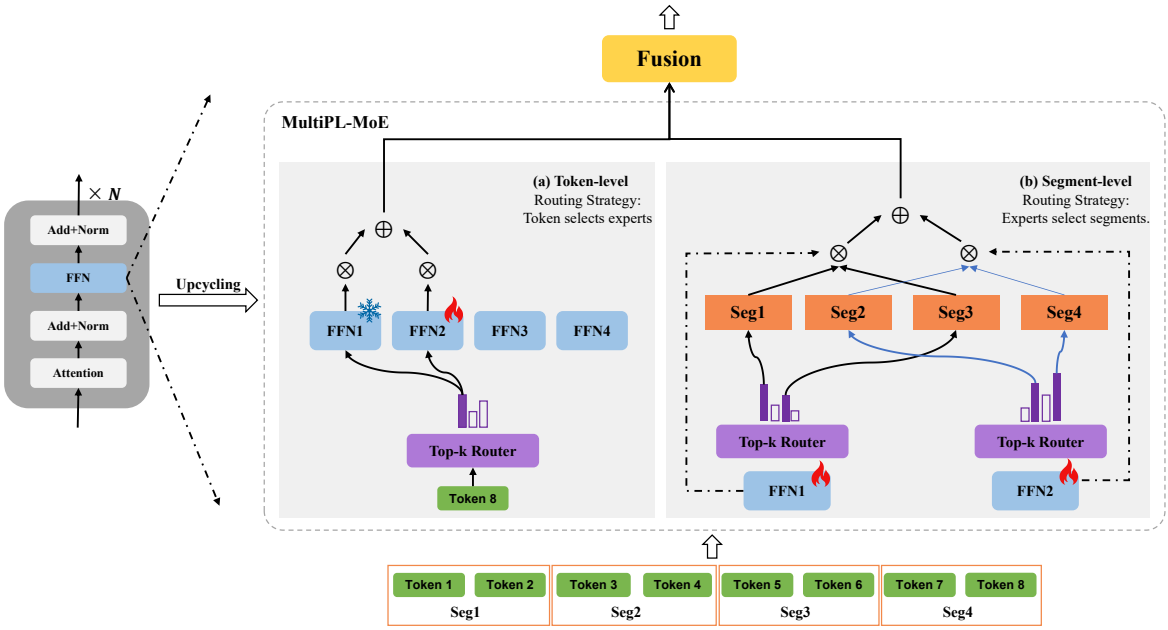

Figure 2: Overall framework of MultiPL-MoE. a)The token-level MoE is a typical MoE structure by upcycling with a shared expert. The routing strategy is token selecting experts. with a novel routing weight normalization method to address scale mismatch.b)The segment-level MoE incorporates two innovative designs to better capture the syntactic structure and contextual patterns of programming languages: 1) Using a sliding window to partition the input token sequence into multiple segments. 2) Adopting a different routing strategy with token-level MoE that allows experts to select the top-k segments.

$\mathcal{B}$ number of samples S. c denotes on average how many experts are utilized by a segment, and N^{seg} is the total number of experts for segment-level MoE.

Three matrices I, D, and U are employed to produce expert-to-segment assignment. The index matrix $I \in \mathbf{1}^{N^{seg} \times r}$ maps segments to experts, where $I[i,j]$ denotes the j-th selected segment by the i-th expert. The weight matrix $D \in \mathbb{R}^{N^{seg} \times r}$ assigns expert weight to the selected segment, while $U \in \mathbb{R}^{N^{seg} \times r \times V}$ is an one-hot version of I used to aggregate segments per expert. To facilitate the computation of affinity scores, we derive the embedding seg'_v for the v-th segment in a batch by averaging the token embeddings within the v-th segment. $SEG'_{\mathcal{B}} = \{seg'_1, \ldots, seg'_v, \ldots, seg'_V\}$ denotes the embeddings of all the V segments within a batch. Following these, the computation procedures of the router and three matrices can be defined as below.

$$\begin{aligned} G_{seg} &= Soft\max(SEG'_{\mathcal{B}} \cdot W_{s\_expert}) \\ D, I &= TopK(G_{seg}, r), U = Onehot(I) \end{aligned} \tag{4}$$

where $G_{seg} \in \mathbb{R}^{N^{seg} \times V}$ denotes the expert-to-segment affinity score matrix. W_{s_expert} denotes the router matrix, and $TopK(\cdot)$ selects the r largest entries for each row of G_{seg}.

Next, we could obtain the input of the i-th expert, denoted as $X_{in}[i] \in \mathbb{R}^{r \times hidden\_size}$, by using the matrix U and the segment embeddings for a batch of input samples $SEG'_{\mathcal{B}} \in \mathbb{R}^{V \times hidden\_size}$. The output of the i-th expert $X_{out}[i]$ is then generated using the expert's FFN transformation. Given X_{out} and the matrices U, D, the output of l-th layer at the segment-level O_l^{seg} could be calculated as below:

$$\begin{aligned} O_l^{seg} &= concat(O_{seg_1}, \ldots, O_{seg_v}, ..., O_{seg_V}) \\ O_{seg_v} &= \sum_{i,j} U[i,j,v] D[i,j] X_{out}[i,j] \\ X_{out} &= FFN_i(X_{in}) \\ X_{in} &= U \cdot SEG'_{\mathcal{B}} \end{aligned} \tag{5}$$

where O_{seg_v} denotes the final output of the v-th segment in a batch across all experts.

Finally, we could fuse the l-th layer outputs of both the token-level and segment-level MoEs:

$$Output^l = W^{tok} \cdot O_{t,l}^{tok} + W^{seg} \cdot O_l^{seg} \tag{6}$$

where W^{tok} and W^{seg} are weight matrices for token-level and segment-level MoEs separately.

3.3 Training Losses.

The hybrid MoE is trained by integrating a next token prediction loss with a load balancing loss, as

described below.

Next Token Prediction Loss. Given T tokens, the post-pretraining next token prediction loss is:

$$L_{NTP} = -\frac{1}{T}\sum_{t=1}^{T}\log p(x_t|Output_{<t}) \quad (7)$$

where x_t is the true token at position t and $Output_{<t}$ represents the output of all tokens before position t.

Load Balancing Loss. Similar to (Zhou et al., 2024), we also incorporate a load balance loss to alleviate the risk of routing collapse in token-level MoE.

$$\begin{aligned} L_{balance} &= \alpha \cdot N^{tok} \cdot \sum_{i=1}^{N^{tok}} f_i \cdot p_i \\ p_i &= \frac{1}{T}\sum_{t\in\mathcal{B}} g_{i,t} \\ f_i &= \frac{1}{KT}\sum_{t\in\mathcal{B}} \mathbf{1}\{\, Token\, t\, select\, expert\, i\} \end{aligned} \quad (8)$$

where α is a hyperparameter that controls the weight of the load balancing loss, K indicates the selected experts, f_i denotes the fraction of tokens dispatched to the i-th expert and p_i denotes the average affinity scores of tokens allocated to the i-th expert.

The final optimization objective is

$$\underset{\theta_{*\_expert}, W_{*\_expert}}{\arg\min} \quad L_{NTP} + L_{balance} \quad (9)$$

where θ_{*_expert} represents the active experts of both token-level and segment-level, and W_{*_expert} denotes the router matrix of both token-level and segment-level.

4 Experiments

4.1 Experiment Setup

Datasets. We select Qwen1.5-1.8B (Team, 2024a) as our backbone model due to its lower computational requirements and ease of upcycling. Our focus is on three low-resource programming languages: Rust, Go, and Ruby, for which Qwen1.5-1.8B exhibits poor performance (see Figure 1). Additionally, we include Python, Java, and C++ as high-resource programming languages to examine their catastrophic forgetting during model updates. We sample 3.3 billion tokens from The Stack (Kocetkov et al., 2022) for post-pretraining, with a 9:1 ratio of low- and high-resource programming languages.

Implementation Details. MultiPL-MoE upcycles Qwen1.5-1.8B with 12 experts per layer, integrating 6 token-level experts (including a shared expert) with top-2 activation and 6 segment-level experts configured as $c = 1, r = 10$. During post-pretraining, the upcycled MoE model updates only the newly introduced experts and the routing mechanism. The training settings include a batch size of 64, a sequence length of 1024, a learning rate of 7e-6 with a cosine scheduler, and an α of load balancing loss of 0.01. We utilize BF16 mixed precision and flash attention (Dao et al., 2022) to speed up the training process. We evaluate on HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) benchmarks extended to multiple programming languages through MultiPL-E adaptation (Cassano et al., 2022).

Baselines. We conduct experiments on several existing baselines on the same dataset to evaluate the efficiency of MultiPL-MoE.

- **Post-pretraining**: Directly perform continue training on Qwen1.5-1.8B.
- **LoRA** (Hu et al., 2022): LoRA settings are applied to all linear layers, and the rank is 8.
- **Gshard** (Lepikhin et al., 2020): In GShard, we set up 12 experts and select the top 3 routing strategy.
- **Mixtral** (Jiang et al., 2024): Each layer employs 8 experts, with per-token routing dynamically selecting and aggregating outputs from the top 2 experts.
- **Qwen1.5-MoE** (Team, 2024b): We upcycle the Qwen1.5-1.8B model to a Qwen-MoE structure with 2.7B activated parameters.

4.2 Experiment Results

Main results and analysis. Table 1 summarizes the pass@1 results of baselines on both low-resource (Rust, Go, Ruby) and high-resource (Python, Java, C++) programming languages under the Humaneval and MBPP benchmarks. The results demonstrate that MultiPL-MoE achieves significant advancements in cross-language generalization, particularly for low-resource programming languages. Compared to MoE architectures, the dense models (Post-pretraining and LoRA) show superior performance on high-resource programming languages across both evaluation benchmarks, while exhibiting degraded effectiveness on low-resource programming languages. MultiPL-

Model	$n_{act-params}$	n_{params}	Programming Languages						Avg.
			Python	Java	C++	Rust	Go	Ruby	
HumanEval									
Qwen1.5	1.8B	1.8B	20.1	10.3	10.0	0.5	2.3	1.1	7.4
Post-pretraining	1.8B	1.8B	**22.8**	10.4	10.0	0.9	2.3	0.6	7.8
LoRA	1.8B	1.8B	20.1	7.2	5.7	0.1	2.1	0.1	5.9
Gshard	3.5B	10.8B	7.9	2.1	7.9	7.3	7.4	6.0	6.4
Mixtral	2.6B	7.5B	8.5	1.6	2.1	0.7	2.9	4.6	3.4
Qwen1.5-MoE	2.7B	14.3B	15.2	9.2	8.6	6.1	8.2	6.5	9.0
MultiPL-MoE	3.5B	10.8B	16.6	**10.6**	**10.6**	**10.9**	**8.7**	**7.1**	**10.8**
MBPP									
Qwen1.5	1.8B	1.8B	**18.0**	19.7	19.4	4.5	8.5	0.2	10.0
Post-pretraining	1.8B	1.8B	4.6	**20.4**	**19.6**	5.1	9.1	0.2	9.8
LoRA	1.8B	1.8B	2.4	12.1	16.9	0.9	6.9	0.3	6.6
Gshard	3.5B	10.8B	0.2	9.0	11.6	11.2	12.6	16.4	10.2
Mixtral	2.6B	7.5B	0.0	9.1	9.6	5.5	10.7	3.9	6.5
Qwen1.5-MoE	2.7B	14.3B	0.0	17.6	18.0	**17.1**	17.1	16.0	14.3
MultiPL-MoE	3.5B	10.8B	1.4	17.3	18.0	16.1	**17.3**	**19.6**	**15.0**

Table 1: Evaluation results across six programming languages on HumanEval and MBPP. Python, Java, C++ are high-resource programming languages. Rust, Go and Ruby are low-resource programming languages. $n_{\text{act-params}}$ is the number of activated model parameters per token, and n_{params} is the total number of model parameters. **Bold** denotes the best, and underline denotes the second-best.

MoE achieves 10.8 pass@1 on HumanEval and 15.0 pass@1 on MBPP with 3.5B activated parameters, greatly surpassing Gshard. MultiPL-MoE and Qwen1.5-MoE achieve consistent performance across both benchmarks, significantly enhancing model capabilities on low-resource programming languages while effectively mitigating catastrophic forgetting in high-resource programming languages.

The results further reveal divergent patterns across benchmarks. MultiPL-MoE achieves state-of-the-art average performance on HumanEval, surpassing the second-best Qwen1.5-MoE by 1.8 points. Notably, our model dominates all programming languages except Python, with Ruby exhibiting the greatest improvement. Although Qwen1.5-MoE is competitive, our model maintains its lead with a higher average pass@1 on MBPP. Post-pretraining performs competitively on MBPP in C++ and Java, but falls short in low-resource languages. We notice that Python exhibits consistently low performance on MBPP across all models except the base model, which is substantially inferior to other languages.

Effect of Key Components. We conduct experiments to evaluate the impact of two critical components in MultiPL-MoE: (1) segment-level MoE and (2) shared expert mechanisms at the token-level. Experiments are carried out using the HumanEval benchmark, comparing pass@1 scores for high- and low-resource programming languages.

Model	HumanEval	
	Original	Expanded
Qwen1.5	40.4	3.9
MultiPL-MoE -w/o segment-level -w/o shared expert	27.3	22.5
MultiPL-MoE -w/o segment-level	30.9	16.8
MultiPL-MoE -w/o shared-expert	33.3	22.3
MultiPL-MoE	**37.8**	**26.7**

Table 2: Performance evaluation on HumanEval, comparing the aggregated pass@1 scores between high-resource (Original) and low-resource (Expanded) programming languages. Dual exclusion (w/o) means the absence of both configurations.

As shown in Table 2, the complete MultiPL-MoE achieves state-of-the-art performance, outperforming the Qwen1.5 on low-resource programming languages by 22.8 points, while exhibiting minor degradation on high-resource programming languages. This trade-off illustrates MultiPL-

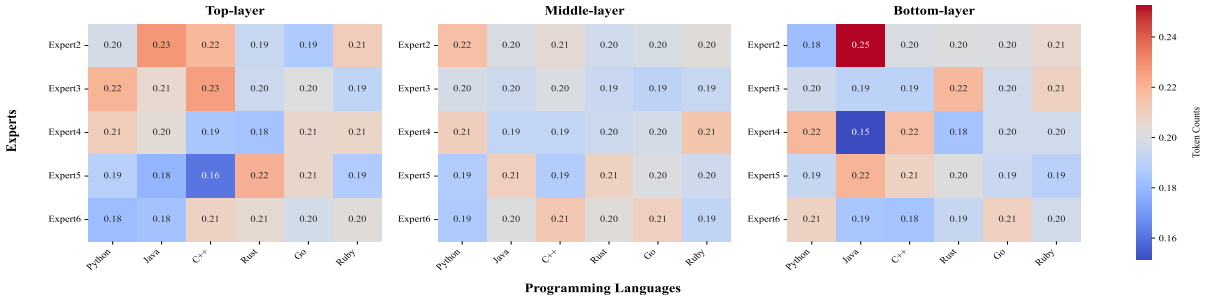

Figure 3: Frequency of selected tokens in different layers. The shared expert (first expert) is omitted from schematic diagram due to its universal application. Our 24-layer architecture spans layer 1 (bottom) to layer 24 (top) with layer 12 as the midpoint.

MoE's capacity to favor cross-lingual generalization while avoiding catastrophic forgetting. Dual exclusion of both components causes severe degradation in high-resource programming languages. The removal of segment-level MoE affects performance for low-resource programming languages, demonstrating its utility. In contrast, the deletion of shared experts has a significant impact on adaptation to high-resource languages, resulting in lower pass@1 scores. These results demonstrate that segment-level MoE is essential for capturing language-specific syntactic patterns in low-resource languages, whereas shared experts stabilize cross-lingual knowledge transfer and mitigate catastrophic forgetting.

4.3 Expert Specialization Analysis

To investigate the specialization patterns of hybrid MoE experts, we examine the selection frequency of experts at the token and segment levels in six programming languages. We compare the frequency with which experts are selected in the top layer (layer 24), middle layer (layer 12), and bottom layer (layer 1) to trace the hierarchical transmission of specialization.

Token-level experts specialization. Figure 3 illustrates the progressive attention to token-level experts over different MoE layers. Specialization patterns emerge among bottom-layer experts, with distinct programming language preferences. Quantitative analysis shows that Expert 2 demonstrates peak Java competency, while Expert 3 shows Rust and Ruby proficiency. Expert 4 maintains balanced Python and C++ engagement, and Expert 6 concentrates exclusively on Python and Go. Middle-layer specialists, on the other hand, display transitional traits, with values clustering primarily around 0.20.

Experts in the top-layer also exhibit distinct specialization patterns divergent from bottom-layer experts, demonstrating the architecture's capacity to aggregate abstract cross-linguistic features. Specifically, Expert 5 specializes in procedural programming paradigms (Rust and Go), while Expert 2 and Expert 3 focus predominantly on object-oriented programming languages. Expert 4 and 6 exhibit no statistically significant linguistic preferences. Notably, Ruby and Go's constantly balanced expert attention across all layers highlights the ongoing difficulty of representing low-resource programming languages, which may result from limited cross-language interchangeability.

Language	Top-2 Selected Segments		
	L1	L12	L24
Python	(10, 3)	(3, 11)	(9, 13)
Java	(12, 5)	(5, 15)	(2, 14)
C++	(14, 10)	(10, 14)	(6, 15)
Rust	(5, 2)	(6, 2)	(11, 15)
Go	(1, 5)	(1, 5)	(16,13)
Ruby	(8, 4)	(8,4)	(9,13)

Table 3: Top-2 selected segments at three layers across six programming languages. ((10,3) represents the segment 10 and 3, which are the most frequently selected by segment-level experts at the corresponding layer. L1 denotes the bottom-layer (layer 1).

Segment-level experts specialization. We investigate segment selection frequencies from segment-level experts across six programming languages, focusing on how language-specific and structural preferences impact experts' specialization. Table 3 demonstrates distinct segment selection patterns. Specially, layer 1 and layer 12 experts exhibit identical top-2 segment activation patterns for C++, Go and Ruby, suggesting cross-linguistic consistency in low-level feature process-

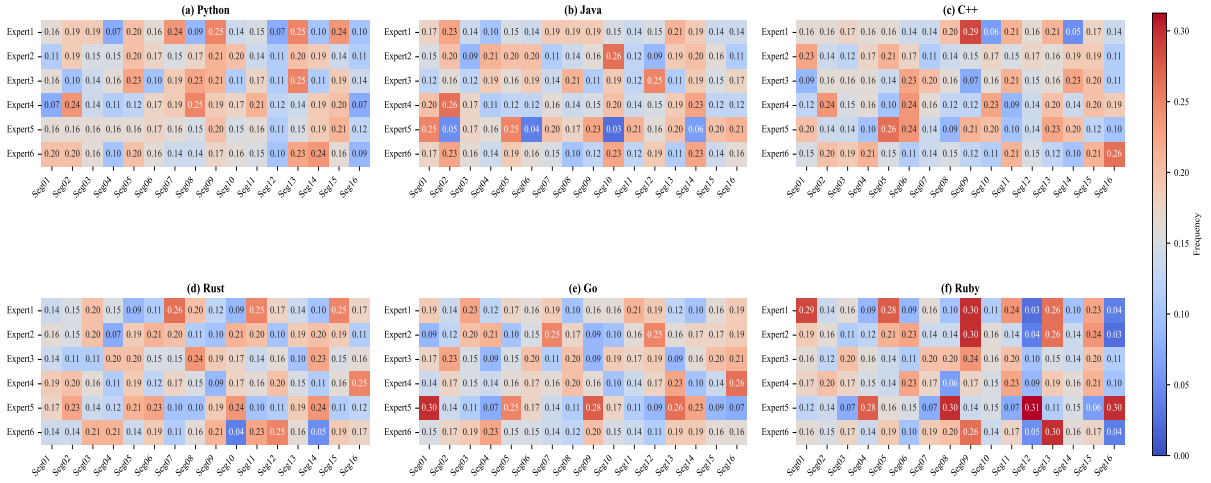

Figure 4: Frequency of selection by segment-level experts in the different programming languages at the top layer (Layer 24) of MoE. Experts at the segment level prefer different segments across six programming languages.

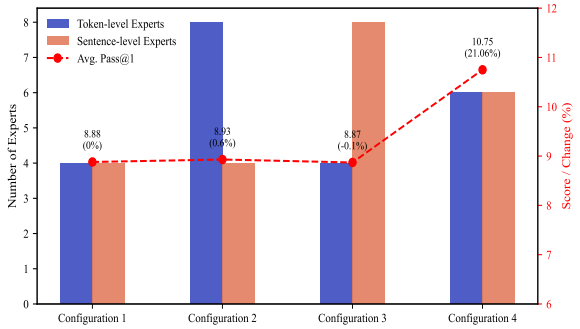

Figure 5: Average performance results for different numbers of experts on HumanEval. The red dashed line denotes the average pass@1 score, computed by averaging pass@1 scores across six programming languages.

ing. Experts in layer 24 exhibit divergent segment preferences.

Figure 4 illustrates that segment-level experts at the top-layer (layer 24) show varied preferences for segments. Experts at the top layer primarily concentrate on segments 5, 9, 12, and 16 across six programming languages, exhibiting cross-linguistic consistency. In particular, Experts appear to prefer segments 5 and 9 in high-resource programming languages (Python, Java, and C++) while engaging more with segments 12 and 16 in low-resource programming languages (Rust, Go, and Ruby). We observe that expert 5 shows distinct preference patterns in different programming languages, excluding Python.

To conduct comparative analysis, we also analyze the segment selection frequencies of experts in layers 1 and 12, see Appendix A.1.

Allocation of Experts. We analyze the relationship between expert configurations and multilingual code reasoning performance. Due to MultiPL-MoE's hybrid two-level MoE design, we investigate four different configurations with varied token-level to segment-level expert ratios (4:4, 8:4, 4:8, and 6:6). These configurations are subsequently referred to as Configuration 1 through 4 in Figure 5 for comparative performance analysis.

As illustrated in Figure 5, Configuration 1 yields a baseline average pass@1 score of 8.88. Expanding token-level expertise while maintaining segment-level capacity of Configuration 2 yields marginal improvement to 8.93, whereas the inverse allocation of Configuration 3 results in slight degradation to 8.87. Notably, the balanced allocation of Configuration 4 achieves the highest average pass@1 score of 10.74, surpassing the Configuration 1 score of 20.91%. The superior performance of Configuration 4 conclusively demonstrates that balanced allocation across two levels is essential for optimal multilingual code reasoning.

5 Conclusion

This paper proposes MultiPL-MoE, a hybrid MoE with both token-level and segment-level. MultiPL-MoE introduces shared experts to capture the commonality of knowledge at the token level, and obtains semantic and logical information between segments at the segment level. Extensive empirical results on two different benchmarks demonstrate the effectiveness of MultiPL-MoE. In summary, MultiPL-MoE is an effective method for extending low-source programming languages in the post-pretraining phase.

6 Limitations

To capture the syntactic structure and contextual patterns of programming languages, we introduce a segment-level MoE with a fixed sliding window mechanism. It would be interesting to explore a dynamic sliding window strategy based on semantic or syntactic code features. Furthermore, while our empirical results validate the effectiveness of MultiPL-MoE, it would be interesting to provide a complete theoretical explanation for the hybrid MoE architecture.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

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

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*.

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

Fan Cui, Chenyang Yin, Kexing Zhou, Youwei Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Yun Liang, Xingcheng Zhang, Demin Song, et al. 2024. Origen: Enhancing rtl code generation with code-to-code augmentation and self-reflection. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9.

Damai Dai, Chengqi Deng, Chenggang Zhao, RX Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y Wu, et al. 2024. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models. *arXiv preprint arXiv:2401.06066*.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359.

Yifeng Ding, Jiawei Liu, Yuxiang Wei, and Lingming Zhang. 2024. XFT: Unlocking the power of code instruction tuning by simply merging upcycled mixture-of-experts. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12941–12955, Bangkok, Thailand. Association for Computational Linguistics.

Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. 2022. Glam: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning*, pages 5547–5569. PMLR.

Mohamad Fakih, Rahul Dharmaji, Yasamin Moghaddas, Gustavo Quiros, Oluwatosin Ogundare, and Mohammad Abdullah Al Faruque. 2024. Llm4plc: Harnessing large language models for verifiable programming of plcs in industrial control systems. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, pages 192–203.

William Fedus, Barret Zoph, and Noam Shazeer. 2022. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39.

Nils Feldhus. 2024. Conversational XAI and explanation dialogues. In *Proceedings of the 20th Workshop of Young Researchers' Roundtable on Spoken Dialogue Systems*, pages 1–4, Kyoto, Japan. Association for Computational Linguistics.

Weimin Fu, Shijie Li, Yifang Zhao, Haocheng Ma, Raj Dutta, Xuan Zhang, Kaichen Yang, Yier Jin, and Xiaolong Guo. 2024. Hardware phi-1.5 b: A large language model encodes hardware domain specific knowledge. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 349–354. IEEE.

Yunhao Gou, Zhili Liu, Kai Chen, Lanqing Hong, Hang Xu, Aoxue Li, Dit-Yan Yeung, James T Kwok, and Yu Zhang. 2023. Mixture of cluster-conditional lora experts for vision-language instruction tuning. *arXiv preprint arXiv:2312.12379*.

Xue Han, Yi-Tong Wang, Jun-Lan Feng, Chao Deng, Zhan-Heng Chen, Yu-An Huang, Hui Su, Lun Hu, and Peng-Wei Hu. 2023. A survey of transformer-based multimodal pre-trained modals. *Neurocomputing*, 515:89–106.

Xue Han, Yitong Wang, Junlan Feng, Qian Hu, Chao Deng, et al. 2025. Loire: Lifelong learning on incremental data via pre-trained language model growth efficiently. In *The Thirteenth International Conference on Learning Representations*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088*.

Sathvik Joel, Jie JW Wu, and Fatemeh H Fard. 2024. A survey on llm-based code generation for low-resource and domain-specific programming languages. *arXiv preprint arXiv:2410.03981*.

Mohammad Abdullah Matin Khan, M Saiful Bari, Do Long, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024. XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6766–6805, Bangkok, Thailand. Association for Computational Linguistics.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*.

Aran Komatsuzaki, Joan Puigcerver, James Lee-Thorp, Carlos Riquelme Ruiz, Basil Mustafa, Joshua Ainslie, Yi Tay, Mostafa Dehghani, and Neil Houlsby. 2022. Sparse upcycling: Training mixture-of-experts from dense checkpoints. *arXiv preprint arXiv:2212.05055*.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*.

Pietro Liguori, Christian Marescalco, Roberto Natella, Vittorio Orbinato, and Luciano Pianese. 2024. The power of words: Generating {PowerShell} attacks from natural language. In *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*, pages 27–43.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. In *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*.

Wamiq Para, Shariq Bhat, Paul Guerrero, Tom Kelly, Niloy Mitra, Leonidas J Guibas, and Peter Wonka. 2021. Sketchgen: Generating constrained cad sketches. *Advances in Neural Information Processing Systems*, 34:5077–5088.

Saurabh Pujar, Luca Buratti, Xiaojie Guo, Nicolas Dupuis, Burn Lewis, Sahil Suneja, Atin Sood, Ganesh Nalawade, Matt Jones, Alessandro Morari, et al. 2023. Automated code generation for information technology tasks in yaml through large language models. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–4. IEEE.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*.

Qwen Team. 2024a. Introducing qwen1.5.

Qwen Team. 2024b. Qwen1.5-moe: Matching 7b model performance with 1/3 activated parameters".

Ali Tehrani, Arijit Bhattacharjee, Le Chen, Nesreen K Ahmed, Amir Yazdanbakhsh, and Ali Jannesari. 2024. Coderosetta: Pushing the boundaries of unsupervised code translation for parallel programming. *Advances in Neural Information Processing Systems*, 37:100965–100999.

Tianwen Wei, Bo Zhu, Liang Zhao, Cheng Cheng, Biye Li, Weiwei Lü, Peng Cheng, Jianhao Zhang, Xiaoyu Zhang, Liang Zeng, et al. 2024. Skywork-moe: A deep dive into training techniques for mixture-of-experts language models. *arXiv preprint arXiv:2406.06563*.

Yeming Wen and Swarat Chaudhuri. 2023. Batched low-rank adaptation of foundation models. *arXiv preprint arXiv:2312.05677*.

Lemeng Wu, Mengchen Liu, Yinpeng Chen, Dongdong Chen, Xiyang Dai, and Lu Yuan. 2022. Residual mixture of experts. *arXiv preprint arXiv:2204.09636*.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*, pages 1–10.

Fuzhao Xue, Zian Zheng, Yao Fu, Jinjie Ni, Zangwei Zheng, Wangchunshu Zhou, and Yang You. Openmoe: An early effort on open mixture-of-experts language models, 2024. *URL https://arxiv.org/abs/2402.01739*.

Yuan Yang, Siheng Xiong, Ali Payani, Ehsan Shareghi, and Faramarz Fekri. 2023. Harnessing the power of large language models for natural language to first-order logic translation. *arXiv preprint arXiv:2305.15541*.

Xinyu Zhang, Siddharth Muralee, Sourag Cherupattamoolayil, and Aravind Machiry. 2024. On the effectiveness of large language models for github workflows. In *Proceedings of the 19th International Conference on Availability, Reliability and Security*, pages 1–14.

Hao Zhou, Zhijun Wang, Shujian Huang, Xin Huang, Xue Han, Junlan Feng, Chao Deng, Weihua Luo, and Jiajun Chen. 2024. Moe-lpr: Multilingual extension of large language models through mixture-of-experts with language priors routing. *arXiv preprint arXiv:2408.11396*.

Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew M Dai, Quoc V Le, James Laudon, et al. 2022. Mixture-of-experts with expert choice routing. *Advances in Neural Information Processing Systems*, 35:7103–7114.

Ming Zhu, Mohimenul Karim, Ismini Lourentzou, and Daphne Yao. 2024a. Semi-supervised code translation overcoming the scarcity of parallel code data. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1545–1556.

Tong Zhu, Xiaoye Qu, Daize Dong, Jiacheng Ruan, Jingqi Tong, Conghui He, and Yu Cheng. 2024b. Llama-moe: Building mixture-of-experts from llama with continual pre-training. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 15913–15923.

Barret Zoph, Irwan Bello, Sameer Kumar, Nan Du, Yanping Huang, Jeff Dean, Noam Shazeer, and William Fedus. 2022. St-moe: Designing stable and transferable sparse expert models. *arXiv preprint arXiv:2202.08906*.

A Appendix

A.1 Comparison of segment frequencies at different MoE layers in six programming languages

By comparing Figure 6, Figure 7, and Figure 4, we conduct comparative analysis from the following two perspectives. From the perspective of model depth, at the shallow level, the segment selections of experts in a single programming language show a high degree of convergence, indicating their universal dependence on the basic grammatical structure. As the number of model layers increases, the expert selection strategies gradually diverge, reflecting the gradual emergence of expert division of labor. From the perspective of cross-language comparison, the segment selections of the same expert in different programming languages vary significantly, indicating that expert decision-making is language-specific. This difference may come from the essential differences between different programming languages in grammatical rules, semantic expressions, and code idioms.

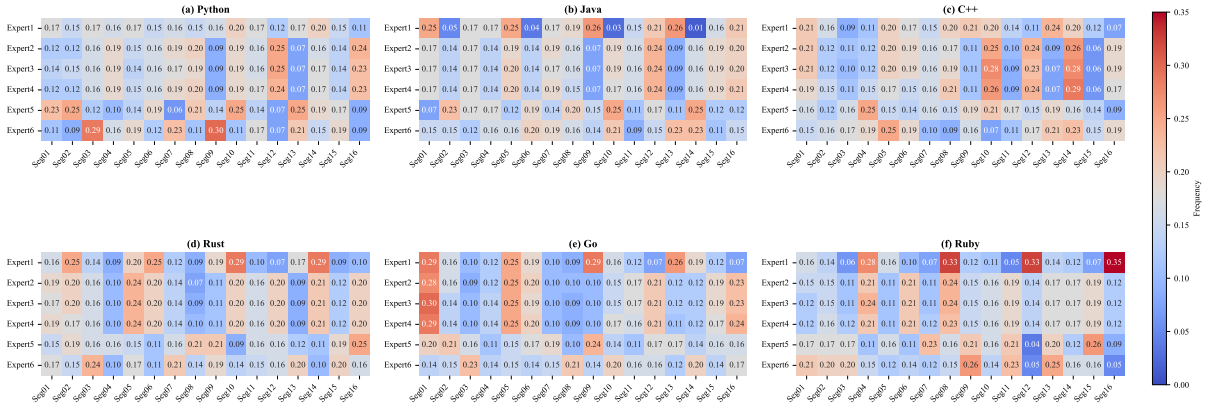

Figure 6: At the bottom-layer of the MoE layer, experts in specific fields have their own unique preferences for different sub-fields of the six programming languages.

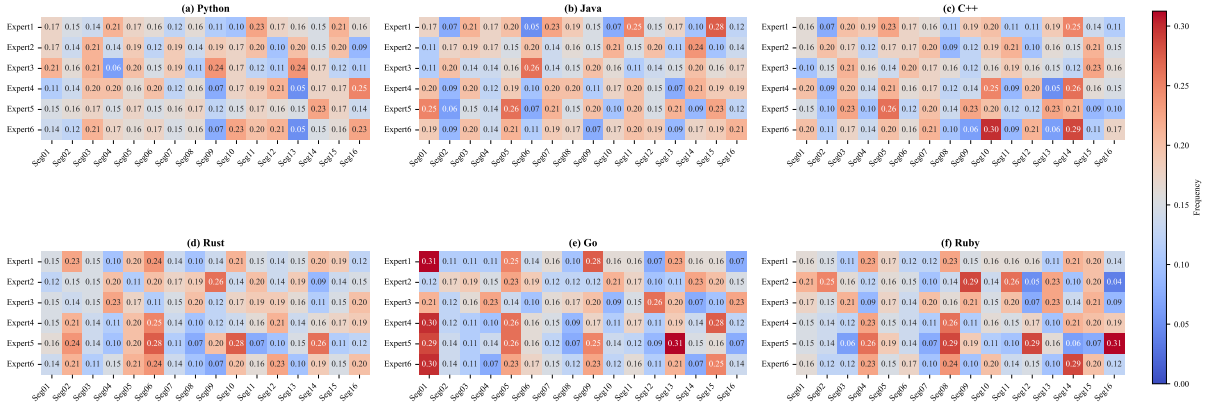

Figure 7: At the middle-layer of the MoE layer, experts in niche areas have their own unique preferences for different niches of six programming languages.