The Journal of Supercomputing (2025) 81:1573
https://doi.org/10.1007/s11227-025-08034-8

A data-augmented model routing framework for efficient LLM deployment in edge–cloud environments

Muhammad Syafiq Mohd Pozi[1] · Yukinori Sato[2]

Received: 6 July 2025 / Accepted: 3 November 2025

Abstract
Large language model (LLM)-based program generation tasks are hindered by high computational demands. These challenges, along with high deployment costs, often pose a barrier to practical applications. To address these, we propose a novel data-augmented multi-LLM model routing approach that classifies prompts based on whether they should be processed on a weak LLM engine or a strong LLM. Experimental results show up to 16 times better efficiency compared to the existing cascaded approaches, while preserving the inference accuracy. Thus, the proposed method optimally allocates prompts across multiple LLMs, reducing computational costs while maintaining inference accuracy.

Keywords Large language model · Program generation · Data augmentation · Routing · Cascading · MEC · Cloud

1 Introduction

The rise of Large Language Models (LLMs) has significantly boosted productivity across various applications and workflows [1]. In programming, tools like Jigsaw [2] and Codex [3] utilize LLMs to translate natural language descriptions into code, while platforms such as GitHub Copilot and Amazon CodeWhisperer assist millions of users in coding more efficiently [4]. By combining LLMs with native code generation for programming languages like Python, developers can automate the process of converting code into executable files for machines [5].

✉ Muhammad Syafiq Mohd Pozi
syafiq.pozi@uum.edu.my

✉ Yukinori Sato
yukinori@cs.tut.ac.jp

1 School of Computing, Universiti Utara Malaysia, UUM Sintok, 06010 Bukit Kayu Hitam, Kedah, Malaysia

2 Department of Computer Science and Engineering, Toyohashi University of Technology, Toyohashi, Aichi 441-8580, Japan

However, the high computational and memory demands of current LLMs often create a bottleneck in program generation, slowing down execution. The efficiency of these models depends largely on their architecture and size. Smaller models, with lower resource requirements, are better suited for deployment on edge devices or in environments like MEC and AI-RAN [6]. These models are lightweight but sacrifice some accuracy. In contrast, larger models, typically hosted in cloud datacenters, can achieve much higher accuracy but come at the cost of significantly higher computational demands [7].

To address these challenges, researchers are exploring hybrid approaches that combine the strengths of both small and large models, aiming to strike a balance between accuracy and resource efficiency [8]. Such approaches hold significant promise for enhancing the performance of LLMs in real-world applications. However, when it comes to program generation tasks, particularly those using a hybrid of small- and large-scale models, a key challenge arises: evaluating the correctness of the generated code.

Unlike LLM applications based on natural language, validating correctness in program generation is more complex, as the generated code must conform to specific syntax, semantics, and application logic. Even small errors in code can cause programs to malfunction, often requiring multiple iterations to correct, which increases computational overhead. For instance, two semantically equivalent code snippets can be vastly different in structure, making traditional metrics, like the BLEU score used in Natural Language Processing, unreliable for assessing program generation tasks [9].

One current solution to this challenge is validating code through actual execution in unit testing. Chen et al. [10] proposed a model cascading approach, where program generation tasks are handled by a hybrid environment of LLM engines. In this approach, a weaker LLM is cascaded to a stronger one until the confidence of the generated output exceeds a predefined threshold. While this method improves accuracy, it is not efficient due to the redundant computations involved in cascading and validating program correctness through unit testing. As a result, this approach can be resource-intensive, negatively impacting performance and overall resource utilization.

To overcome these limitations, we propose a data-augmented routing method for program generation tasks that leverages heterogeneous LLM engines deployed across cloud environments and regional MEC (Multi-access Edge Computing) servers. First, we introduce a mask-based data augmentation strategy that enriches the prompt dataset with varying difficulty levels, improving the router's generalization capability and robustness to unseen inputs. Second, we develop a supervised routing framework that fine-tunes a lightweight pretrained language model to classify prompt difficulty and route tasks accordingly—either to a lightweight LLM on a nearby MEC server or a more capable LLM in the centralized cloud. This design integrates data-level enhancement with model-level optimization, reducing redundant computation while maintaining high program generation accuracy.

This paper is organized as follows: Sect. 2 reviews the existing works on program generation using LLMs. Section 3 introduces our hybrid LLM architecture based on a model routing approach. Section 4 presents experimental comparisons between our routing-based approach and existing techniques. Section 5 discusses the experimental results, and Sect. 6 concludes the paper.

2 The existing works of program generation tasks

Leveraging heterogeneous LLM environments—comprising both weak and strong models—has emerged as a promising way to reduce response latency while preserving code generation quality. Two main strategies have been explored: cascading and routing. The cascading strategy sequentially escalates a task to stronger models based on confidence evaluation, while the routing strategy predicts the most appropriate model for each prompt in advance, avoiding redundant computations. We discuss each in detail below.

2.1 Cascading-based strategy

The cascading strategy triggers a stronger model only when a weaker model fails to meet a predefined confidence threshold. This mechanism, well-established in domains like image classification [11], has been adapted for program generation using LLMs.

In this context, a weak LLM first generates a candidate program. If the output passes predefined self-evaluation checks—typically using auto-generated unit tests—it is accepted. If not, the task is forwarded to a more capable LLM, typically deployed in a cloud data center, for regeneration [10]. This selective escalation helps reduce reliance on costly cloud models while ensuring output correctness.

Figure 1 illustrates this flow. The initial LLM attempts the task; if its result passes unit testing, it is finalized. Otherwise, the prompt moves to the next model in the cascade. This process repeats until the result meets the quality criteria or reaches the final model.

A key distinction in program generation is its binary evaluation—code either passes or fails unit tests—unlike typical NLP tasks that tolerate partial correctness. To exploit this, recent methods generate both code and corresponding unit tests, running them on the same inference server [10]. While effective, the main issue that it adds significant runtime cost due to repeated program execution and validation.

2.2 Routing-based strategy

In contrast, routing-based strategies predict in advance which model is best suited for a given prompt. Inference is then performed on a single model, avoiding the cumulative cost of cascading.

The router—often a lightweight predictor trained via regression or classification—estimates task difficulty and selects between a weak or strong LLM [12]. This makes routing efficient for runtime and resource usage.

Another type of routing is found in Mixture-of-Experts (MoE) architectures [13], which combine multiple LLMs to perform tasks such as question answering.

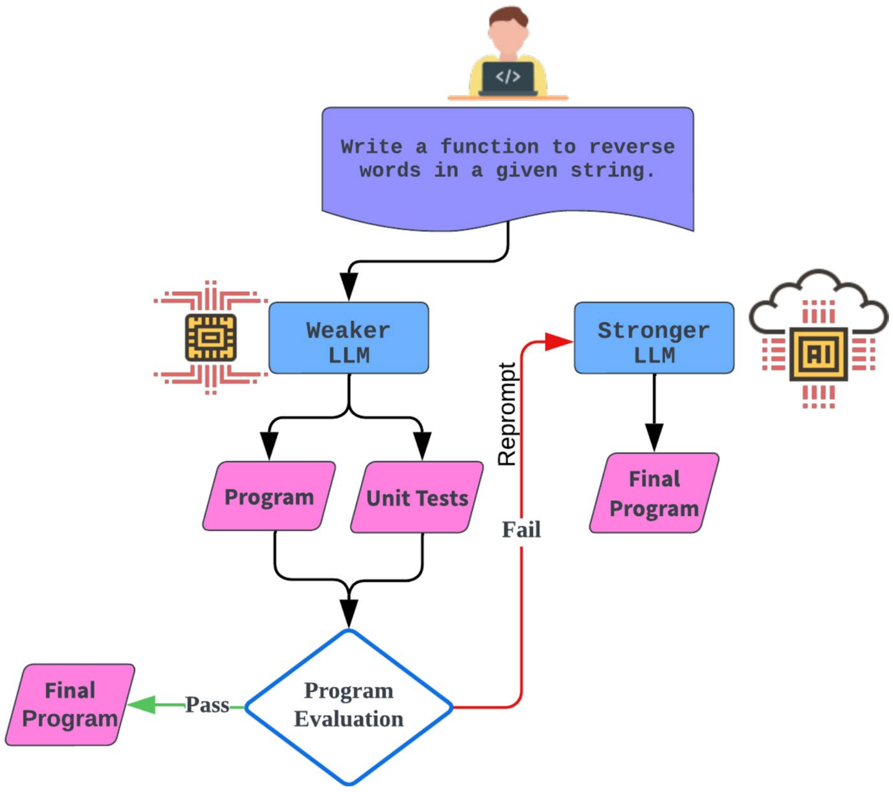

Fig. 1 Cascading strategy. If the generated program receives a negative evaluation, the query will be reprompted by a more advanced LLM for improved results

However, MoE models often require task-specific tuning and assume internal access to all expert models, which limits their flexibility when integrating closed-source LLMs.

As shown in Fig. 2, the routing-based design uses a lightweight model to analyze incoming prompts and route them accordingly. For simple prompts, weak LLMs deployed on MEC or edge devices handle inference. For complex prompts, routing directs them to stronger LLMs in the cloud—achieving cost-effective and accurate program generation.

3 Data augmented routing-based methodology for program generation task

This section describes the methodology used to design, train, and evaluate the proposed routing framework. The approach is organized into several key stages. It begins with a Supervised Learning Framework for LLM Behavior Analysis, which

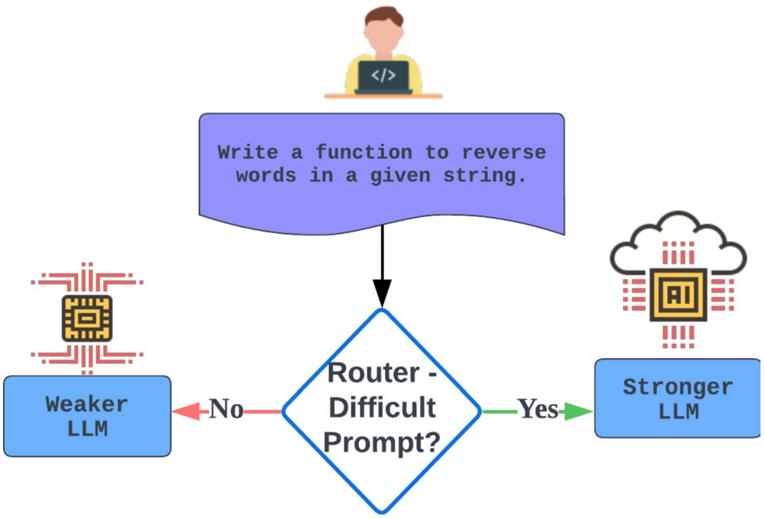
Write a function to reverse words in a given string.
Router - Difficult Prompt?
No
Yes
Weaker LLM
Stronger LLM

Fig. 2 Routing strategy. A router will determine whether the prompt is processed by a weaker LLM or a stronger LLM

models the relationship between prompts and LLM outputs to enable interpretable behavioral assessment. Then, Dataset Preparation and Prompt Annotation outline how prompts, test cases, and evaluation labels are constructed. The Data Augmentation stage enhances the dataset with diverse prompt variations to improve model generalization, followed by Router Training, which fine-tunes a lightweight model to classify prompt difficulty and guide routing decisions. Finally, the Offloading Mechanism describes how the trained router dynamically assigns tasks between weaker and stronger LLMs to optimize both accuracy and computational efficiency.

3.1 Supervised learning framework for LLM behavior analysis

To understand the limitations of a Large Language Model (LLM), we can adopt a supervised learning perspective, where the prompt serves as the input X, and the corresponding LLM output is treated as the target Y. By learning this input–output mapping, we can approximate and analyze the LLM’s behavior using a smaller, more interpretable surrogate model (e.g., a classifier). Such a surrogate model, being significantly less complex, may fail to replicate the full reasoning capacity of the original LLM—highlighting specific limitations such as difficulties in advanced reasoning or program synthesis tasks.

3.2 Dataset preparation and prompt annotation

Two datasets are used in this study to evaluate our method: HumanEval and Mostly Basic Python Problems (MBPP) [14]. Table 1 summarizes the dataset used in this work.

A dataset, D, consists of a user prompt p and a predefined test case u, which includes the input for the program and its expected output, based on the program generated by the LLM in response to the prompt p. Tables 2 and 3 describe example prompts, each paired with a predefined test case u.

Each dataset contains user prompts that are used to generate respective programs by querying the prompt to the selected LLM. The generated programs are evaluated through the test case u, and annotated with y, where y is a *pass* or *fail* status. Thus, every prompt in the dataset has an associated $< p >:< u >:< y >$ relationship. Figure 3 illustrates the data annotation process.

In program generation tasks, each user prompt is labeled with either a "pass" or "fail" status based on the performance of a weaker LLM in generating correct programs. For each prompt p, five programs are generated by the weaker LLM and evaluated against a test case u. If all five programs produce the expected output, the prompt is annotated as a "pass" (i.e., $y = 1$). If fewer than five succeed, the prompt is labeled as a "fail" (i.e., $y = 0$).

This consistency-based evaluation accounts for the inherent non-determinism of LLM outputs and provides a stable and reliable success signal. By adopting a strict 5/5 all-pass rule, only prompts that consistently yield correct programs across multiple generations are labeled as “pass.” This reduces label noise and prevents ambiguous cases (e.g., 3/5 or 4/5 partial success) that may not reflect true capability. Such a criterion enables the annotation process to more accurately capture the weaker LLM’s limitations and ensures the router is trained on well-differentiated examples. Examples of annotated prompts are shown in Table 4.

Our proposed router for selecting the weaker or stronger LLM engine relies on two key components: data augmentation and a classification model. Data augmentation generates variations of existing data to enhance training diversity, while the classification model uses the augmented data to identify prompts suitable for the weaker LLM.

3.3 Component 1: data augmentation

Our primary contribution is a data augmentation method designed to enhance the router’s generalization capability by increasing dataset variability. This method comprises two key processes: Masking and Fill-Mask. Figure 4 illustrates the workflow of the proposed data augmentation technique, providing a visual representation of how masking and fill-mask processes modify the original dataset. Algorithm 1 then provides a detailed step-by-step description of the augmentation process, outlining

Table 1 Comparison of HumanEval and MBPP datasets

Dataset	Purpose	# Problems
HumanEval	Evaluating code generation models	164
MBPP	Benchmarking programming tasks	378

Table 2 Example of prompts and test cases from the HumanEval dataset

Prompt (p)	Test case (u)
Write a function to check if a string is a palindrome	Input: "racecar"
	Expected Output: True
Write a function to find the smallest positive integer not present in a list	Input: [3, 4, −1, 1]
	Expected Output: 2
Write a function to compute the factorial of a number	Input: 5
	Expected Output: 120

Table 3 Example of prompts and predefined test cases from the MBPP dataset

Prompt (p)	Test case (u)
Write a function to check if a number is prime	Input: 11
	Expected Output: True
Write a function to reverse words in a given string	Input: "program Python"
	Expected Output: "Python program"
Write a function to find the n-th Fibonacci number	Input: 6
	Expected Output: 8
Write a function to calculate the sum of a list of numbers	Input: [1, 2, 3, 4, 5]
	Expected Output: 15

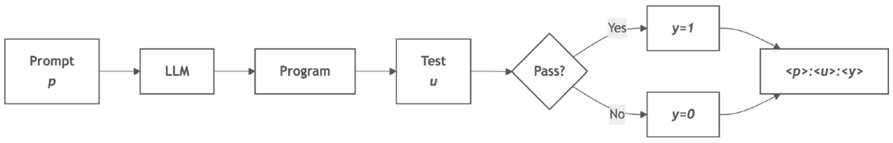

Fig. 3 LLM-generated programs from prompts p are tested against cases u and annotated as $y = 1$ (pass) or $y = 0$ (fail), producing evaluative $< p >:< u >:< y >$ records

how tokens are selected for masking, replaced with [MASK], and subsequently predicted using an LLM.

Algorithm 1 Data Augmentation with Masking and Fill-Mask

```
1: Input: Dataset $P = \{p_1, p_2, \ldots, p_n\}$, Mask probability $p_m = 0.10$, Pretrained LLM model
   $M$.
2: Output: Augmented dataset $P_f$.
3: Initialize $P' \leftarrow \emptyset$
4: for each $p_i \in P$ do
5:     Tokenize $p_i$ into a sequence of tokens $T = [t_1, t_2, ..., t_m]$
6:     // Step 1: Apply Masking
7:     for each token $t_j \in T$ do
8:         Generate a random number $r$ in range $[0, 1]$
9:         if $r \leq p_m$ then
10:            Replace $t_j$ with [MASK]
11:        end if
12:    end for
13:    Masked text: $p'_i \leftarrow T$
14:    // Step 2: Fill Mask using LLM
15:    for each [MASK] token in $p'_i$ do
16:        Predict token using model $M$:
17:            $\hat{t} \leftarrow \arg\max P(v \mid p'_i, M)$ where $v$ is a candidate word
18:        Replace [MASK] with $\hat{t}$
19:    end for
20:    Augmented text: $p''_i \leftarrow p'_i$
21:    $P' \leftarrow P' \cup \{p''_i\}$
22: end for
23: $P_f \leftarrow P \cup P'$ return $P_f$
```

Algorithm 1 performs prompt augmentation by masking a subset of tokens in each prompt from the input dataset, predicting the top k possible tokens for each masked position using a language model, and then filling the masked tokens with these predictions. For each prompt, a masked version is created, and the language model generates possible completions for the masked positions. The language model used for filling the mask is known as GraphCodeBERT [15], as tabulated in Table 6. Then, the filled prompts are added to a new augmented dataset.

After processing all prompts, the final augmented dataset is formed by combining the original dataset with high-quality augmented prompts. The result is an enriched dataset, P_f, which is returned for further use.

The parameters specified for Component 1 are tabulated in Table 5. The parameters for Component 1's parameters consist only of the probability of masking, which determines the likelihood of masking elements in a string, and the augmented size relative to the original dataset $\mathbf{X}$.

Table 4 Prompts annotated with success status indicating whether the weaker LLM engine generate correct (pass) or incorrect (fail) program

Prompt (p)	Status (y)
Write a function to check if a number is prime	Fail
Write a function to reverse words in a given string	Pass

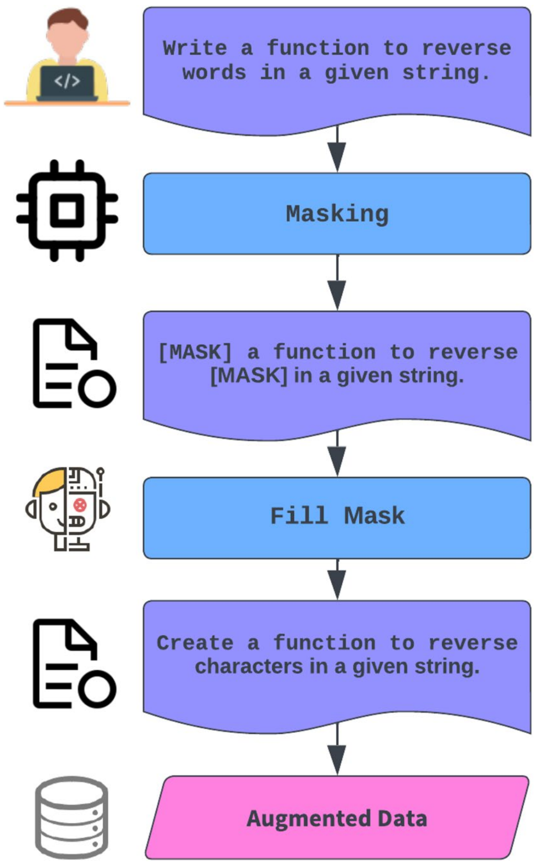
Write a function to reverse words in a given string.
Masking
[MASK] a function to reverse [MASK] in a given string.
Fill Mask
Create a function to reverse characters in a given string.
Augmented Data

Fig. 4 Proposed data augmentation technique aims to create synthetic data to enhance dataset variability. The process begins by randomly masking parts of the given prompt and then filling in the masked sections with different tokens predicted by a specified language model

3.4 Component 2: router training

In Component 2, the router is trained through a supervised learning approach, which is to predict whether the prompt can be inferred at weaker LLM or not. Note that, the data have been annotated such as described in Sect. 3.2.

Let $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$ represents the input text data, where each $\mathbf{x}_i$ is a token sequence of prompt $p_i \in P$. Also, let $\mathbf{y} = \{y_1, y_2, \dots, y_n\}$, represent the corresponding status of each p_i.

Thus, we perform fine-tuning process on an existing pretrained language model: CodeBERT-python [16]. The fine-tuning process is detailed in Algorithm 2.

Algorithm 2 Fine-Tuning a Language Model with Fill-Mask

Table 5 Parameters for the data augmented routing-based methodology

	Component 1	Component 2
Task	Fill mask	Classification
Pre. Model	GraphCodeBERT [15]	CodeBERT-python [16]
Prob. Mask	0.10, 0.15, 0.20	–
Aug. size	$1 \times \text{Size}(\mathbf{X})$	–
Batch size	–	64
Epoch	–	200
# Classes	–	2
Input Length	–	≤512
Optimizer	–	Algorithm: SGD+Nesterov [17]
		Learning rate: 0.001
		Momentum: 0.95
Loss function	–	Cross entropy loss

```
1: Input: Pretrained model $M_{\text{pre}}$, Dataset $\mathbf{X} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$, Hyperparameters $\theta = \{\alpha, \beta, \text{epochs}\}$
2: Output: Fine-tuned model $M_{\text{ft}}$
3: $M_{\text{ft}} \leftarrow M_{\text{pre}}$
4: Split dataset $\mathbf{X}$ into $\mathbf{X}_{\text{train}}, \mathbf{X}_{\text{val}}, \mathbf{X}_{\text{test}}$ such that $\mathbf{X}_{\text{train}} \cup \mathbf{X}_{\text{val}} \cup \mathbf{X}_{\text{test}} = \mathbf{X}$
5: $M_{\text{ft}} \leftarrow M_{\text{ft}} + \text{ClassificationHead}(n)$        ▷ where $n$ is the number of classes
6: $(\mathbf{X}_{\text{train}}, A) \leftarrow \text{Tokenizer}(\mathbf{X}_{\text{train}})$        ▷ where $A$ is the attention mask
7: $(\mathbf{X}_{\text{train}}, \mathbf{y}_{\text{mask}}) \leftarrow \text{FillMaskProcess}(\mathbf{X}_{\text{train}})$        ▷ Algorithm 1
8: Set training parameters: $\theta = \{\alpha, \beta, \text{epochs}\}$
9: for epoch $e \in \{1, \ldots, \text{epochs}\}$ do
10:     for each batch $(\mathbf{x}_{\text{batch}}, \mathbf{y}_{\text{mask}}) \in \mathbf{X}_{\text{train}}$ do
11:         $\hat{\mathbf{y}} \leftarrow M_{\text{ft}}(\mathbf{x}_{\text{batch}}, A_{\text{batch}})$
12:         $\mathcal{L} \leftarrow \text{LossFunction}(\hat{\mathbf{y}}, \mathbf{y}_{\text{mask}})$
13:         Optimize $M_{\text{ft}}$ using $\alpha$ and $\mathcal{L}$
14:     end for
15: end for
16: Monitor $\mathcal{L}_{\text{val}}$ on $\mathbf{X}_{\text{val}}$
17: if $\mathcal{L}_{\text{val}}$ does not improve then
18:     Adjust $\theta$
19: end if
20: $\mathcal{P} \leftarrow \text{Evaluate}(M_{\text{ft}}, \mathbf{X}_{\text{test}})$
21: Save $M_{\text{ft}}$
```

Algorithm 2 fine-tunes a pretrained language model (M_{pre}) to classify whether a prompt can be inferred by a weaker LLM. It begins by adding a classification head and splitting the dataset into training, validation, and test sets. The input sequences are tokenized and augmented using a fill-mask process. During training, the model iterates over batches, computing predictions, loss, and updates to optimize parameters. Validation loss is monitored, and hyperparameters are adjusted if needed. Finally, the fine-tuned model (M_{ft}) is evaluated on the test set and saved for deployment.

The parameters specified for Component 2 are tabulated in Table 5. For Component 2, the training batch size is set to 64, slightly above the typical range (e.g., 8–16). This allows the model to process more samples simultaneously, leading to more stable gradient updates and improved convergence.[1] The number of classes is set to 2, indicating whether the prompt should be offloaded to either a weaker or stronger LLM model.

Note that the maximum input length is 512 tokens. This is because the architecture limitation that is used as a router can only accept 512 tokens and below. In this case, input that has length more than 512 tokens will not be included in the training process.

Finally, the loss function, $\mathcal{L}$ for Component 2 is a Cross Entropy Loss, expressed such as follows:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(p_i) + (1 - y_i)\log(1 - p_i)\right) \tag{1}$$

where N is the number of samples. y_i is the number of samples i (0 or 1). p_i is the predicted probability that the sample belongs to the positive class.

Here, we specify the parameters used for both Component 1 and Component 2, describing their roles in data augmentation and classification processes.

3.5 Offloading mechanism

Due to the limitation of language model, the Router is trained only on prompts with length of 512 tokens or fewer. Therefore, if any given prompt, p, exceeds the length of 512 tokens, it will bypass the Router and be re-directed straight to the stronger LLM. Otherwise, a routing inference will occur to determine whether the weaker LLM can successfully process the prompt. Figure 5 illustrates the offloading procedure.

4 Experimental findings

4.1 Parameters and evaluation modeling

Table 6 presents the parameters for the experiment evaluating our method. We compare the proposed routing method against the cascading approach described in [10] (also referred to Fig. 1). In the cascading method, a code completion prompt is sent to the large language model (LLM) to generate the correct program and a set of self-generated k-unit tests to evaluate the program. These test results are then used to determine the cascading threshold.

Llama–3.1-8B is used as the weaker LLM engine, while GPT-4o Mini serves as the stronger engine. The weaker engine is deployed in an Ollama container, with access to an Intel® Xeon® W-2102 CPU, an NVIDIA RTX A4000 GPU, 24GB

[1] Trained on NVIDIA A100 with 80GB of VRAM.

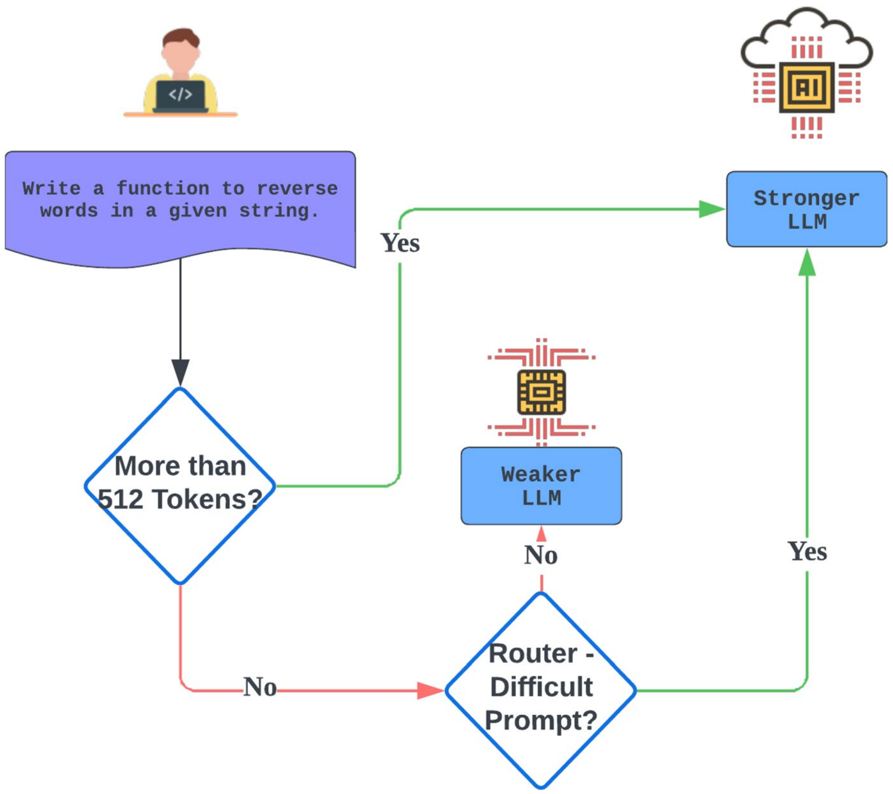
Write a function to reverse words in a given string.
Yes
Stronger LLM
More than 512 Tokens?
Weaker LLM
No
Yes
No
Router - Difficult Prompt?

Fig. 5 LLM routing strategy—Prompts over 512 tokens are sent to a strong cloud LLM. Shorter prompts (more than 512 tokens) are classified by a "Router" as easy or difficult; easy prompts go to a weaker LLM, difficult ones to a stronger LLM

Table 6 Parameters for both offloading mechanisms

Item	Cascading	Routing
LLM	Llama−3.1-8B (MEC)	Llama−3.1-8B (MEC)
	GPT-4o Mini (Cloud)	GPT-4o Mini (Cloud)
		CodeBERT (as a router at MEC)
		- Comp. 1: GraphCodeBERT [15]
		- Comp. 2: CodeBERT-python [16]
Dataset	5 Sets of Solutions from Llama−3.1-8B	
	5 Sets of Solutions from GPT-4o Mini	
Program Evaluation	7-Unit Tests	5-Fold Cross-Validation

of RAM, and 512GB of SSD storage. Moreover, our routing method also requires an additional language model, which serves as a foundation for learning the limitations of the weaker LLM engine. CodeBERT [15, 16, 18] has been selected for this purpose due to its ability to effectively understand the complexities of programming tasks.

Either way, for each problem in each dataset, we generated 5 sets of solutions from the weaker LLM and 5 sets of solutions from stronger LLM (as in Table 6). Hence, the probability for LLM to generate correct program is such as follows:

$$P_w(+) = \frac{C_w}{N_w} \tag{2}$$

$$P_s(+) = \frac{C_s}{N_s} \tag{3}$$

where C_w and C_s is the number of correct programs generated by weaker and stronger LLM, respectively, and N_w and N_s is the total number of programs generated by the weaker and stronger LLM as a final program, respectively.

While employing different evaluation paradigms, both approaches yield directly comparable accuracy metrics. Cascading deterministically verifies each prompt against seven unit tests (accuracy = compliant prompts/total),while Routing's 5-fold cross-validation (CV)—a technique for assessing model generalizability by partitioning data into five subsets—provides mean accuracy across all prompt predictions. Figure 6 demonstrates how both achieve complete file evaluation, ensuring methodological validity despite their differing approaches to correctness assessment.

In addition, program accuracy is assessed using the EvalPlus benchmark tool [14], which determines whether generated programs produce the correct expected outputs defined by each dataset. The evaluation uses the *pass@1* metric, a special case of the general *pass@e* formulation:

$$\text{pass@}e = 1 - \prod_{i=1}^{N} \left(1 - \frac{C_i}{e}\right) \tag{4}$$

where N is the number of problems, C_i is the number of correct programs for problem i, and e is the number of samples per problem. In this work, we fix $e = 1$, meaning the model is allowed only one attempt per problem.

The pass@1 metric is further decomposed into platform-specific accuracy scores (H), shown in Table 7, which empirically reflect the success rates of program generation across different deployment platforms (MEC and Cloud). These H values represent empirical estimates of pass@1 for each platform:

Here, $P_w(+)$ and $P_s(+)$ indicate correct programs generated by the weaker and stronger LLMs, respectively, while N_w and N_s are the number of problems handled by each platform. The indicator function $\mathbf{1}_{y=c}$ routes problems based on predicted difficulty. The platform-specific metrics H thus reflect both the accuracy of the deployed LLMs and the effectiveness of the offloading mechanism under the pass@1 setting.

Based on Figs. 7 and 8, the program accuracy H, produced by MEC-only method is the lowest for both datasets, which is 53% for HumanEval dataset and 65% for MBPP dataset. Notably, the accuracy of the generated program by the Cloud-only

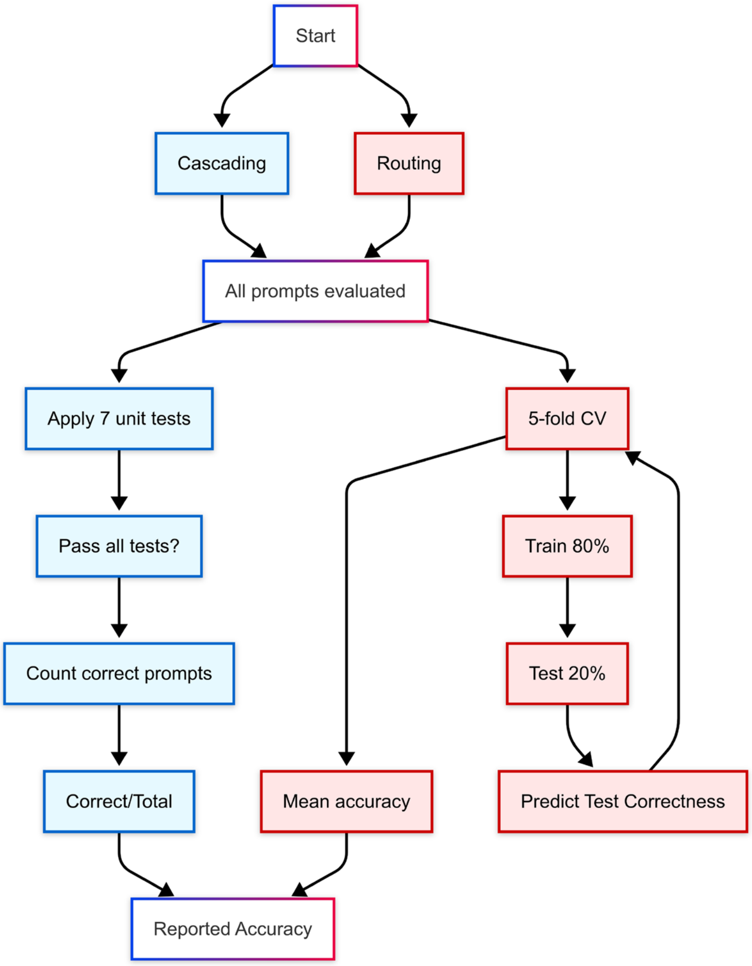

Fig. 6 Evaluation workflow comparison showing how both approaches achieve comparable prompt-level accuracy assessment. While methodologies differ (deterministic unit tests vs. statistical cross-validation), all prompts are evaluated exactly once in both frameworks, enabling fair performance comparison

Table 7 Platform-specific program accuracy (H) based on pass@1 evaluation

Platform	Program accuracy: H
MEC	$H_{\text{MEC}} = \frac{\sum_{i=1}^{N} P_w(+)}{N_w}$
Cloud	$H_{\text{Cloud}} = \frac{\sum_{i=1}^{N} P_s(+)}{N_s}$
Overall	$H_{\text{Overall}} = \frac{\sum_{i=1}^{N} \left(P_w(+)\cdot\mathbf{1}_{y=1} + P_s(+)\cdot\mathbf{1}_{y=0}\right)}{N}$

method is the highest for both datasets, which is 85% for HumanEval dataset and

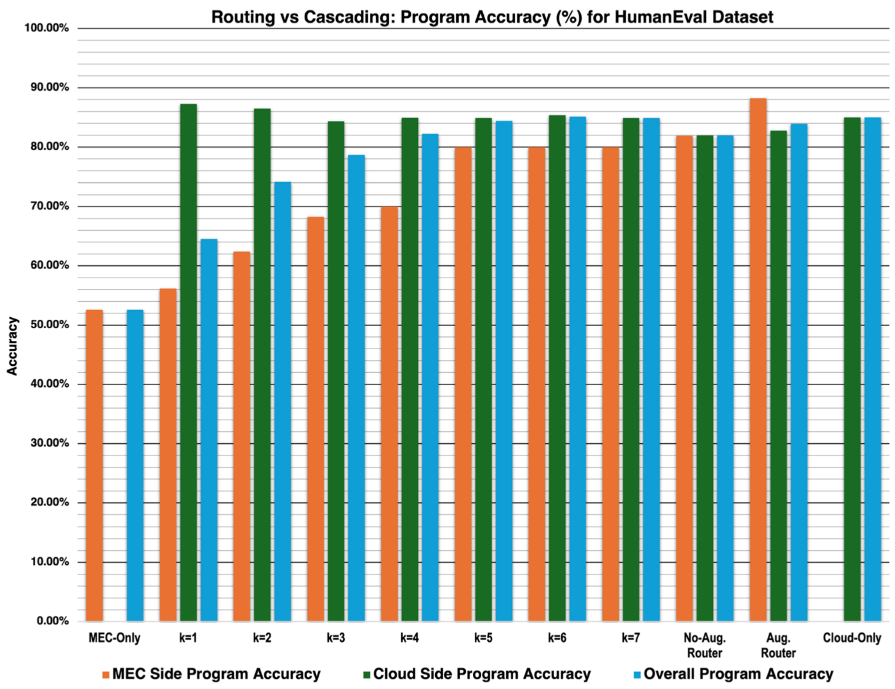

Fig. 7 Program accuracy $H(\%)$, based on HumanEval dataset with respect to each offloading approach

83% for MBPP dataset.

For cascading-based solutions, increasing the number of unit tests to evaluate the program generated by weaker LLM (MEC) improves the overall program accuracy, as well as accuracy in generating program at MEC and Cloud.

Routing-based solutions show a significant program generation accuracy improvement in the edge-cloud continuum environment. For the HumanEval dataset, both No-Aug. Router and Aug. Router achieved program generation accuracy's of 82% and 84%, respectively. A similar trend is observed with the MBPP dataset, where No-Aug. Router and Aug. Router attained accuracy's of 75% and 78%, respectively. For both datasets, Aug-Router outperforms No-Aug-Router in terms of program generation accuracy.

4.2 Program generation runtime

The program generation time (in seconds) is measured from the beginning of prompt evaluation to the completion of final program generation. This measurement focuses on the computational aspects of generation within the LLM pipeline, excluding network transmission or communication latency typically associated with edge-cloud offloading. A detailed breakdown is provided in Table 8.

Cascading-based approach requires 3 measurements, while routing-based approach only requires 2 measurements. Figures 9 and 10 illustrates the program

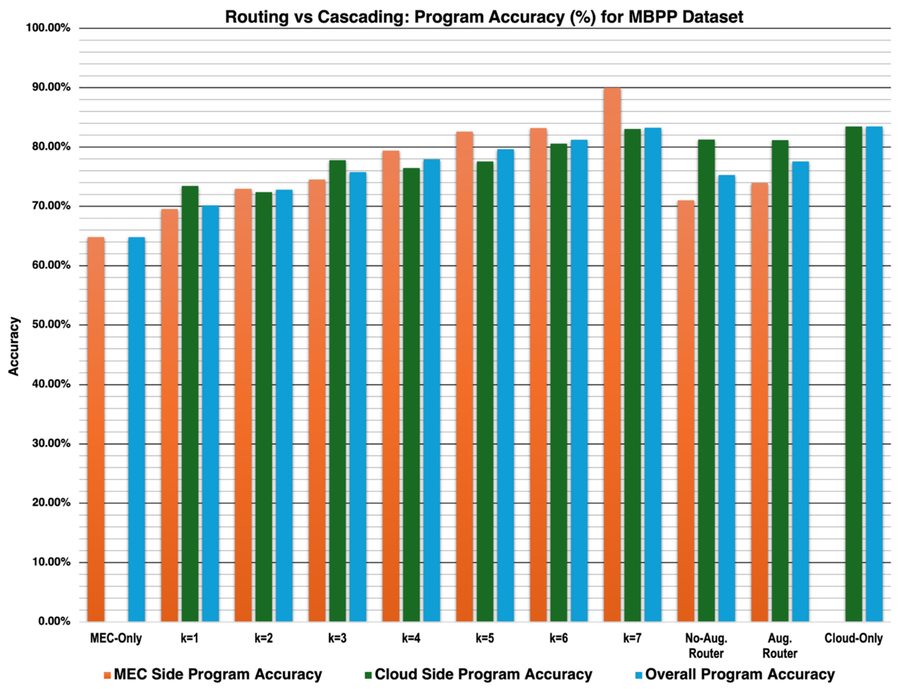

Fig. 8 Program accuracy, $H(\%)$, based on MBPP dataset with respect to each offloading approach

generation runtime measurements for both approaches. The cascading approach results in significantly longer program generation times, taking 16 times longer than the routing-based approach.

Finally, Table 9 presents the routing and accuracy rates for both the HumanEval and MBPP datasets under two configurations: without data augmentation (No-Aug) and with data augmentation (Aug). The routing rate represents the proportion of prompts correctly directed to the appropriate LLM engine (edge or cloud), while the accuracy rate at MEC indicates the success rate of program generation when prompts are processed by the weaker LLM at the MEC side.

The results show that data augmentation significantly enhances the router's discriminative capability. The augmented router more effectively distinguishes difficult prompts and routes them to the stronger cloud model, thereby reducing the number of misrouted complex cases. As a result, the augmented configuration achieves

Table 8 Program generation runtime measurement for cascading and routing methods in seconds (s)

Measurement	Cascading	Routing
Prompt evaluation duration		✓
Prompt inference duration	✓	✓
Unit test generation duration	✓	
Unit test evaluation duration	✓	

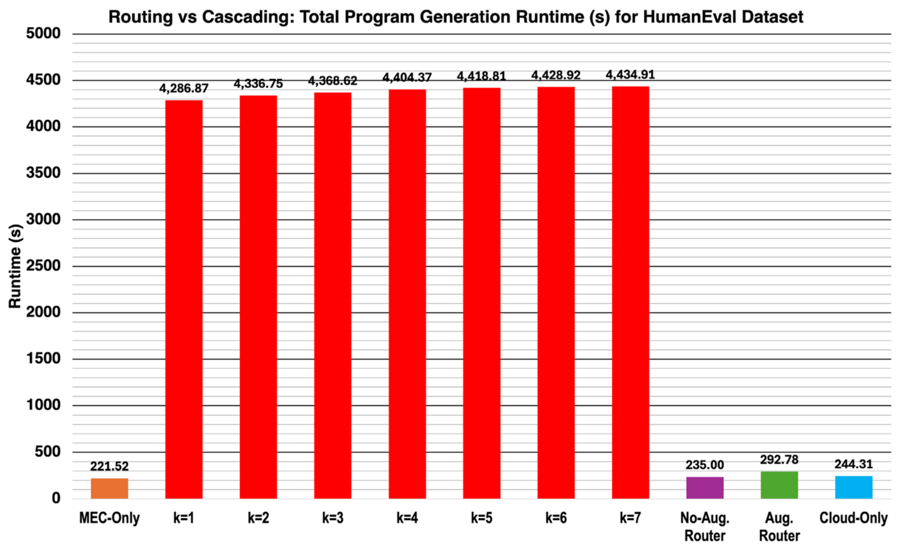

Fig. 9 Total program generation runtime (s) based on HumanEval dataset with respect to each offloading approach

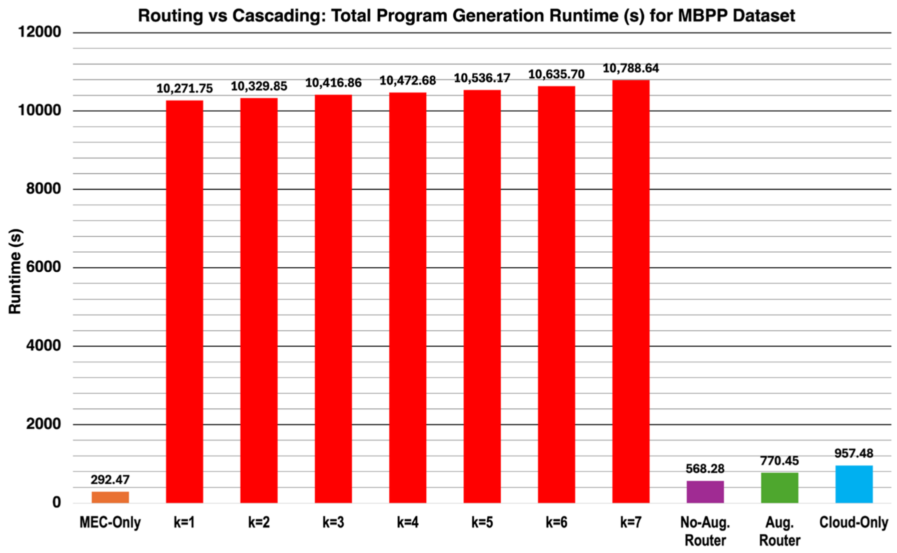

Fig. 10 Total program generation runtime (s) based on MBPP dataset with respect to each offloading approach

higher routing accuracy (from 74.39 to 79.27% on HumanEval and from 41.80 to 50.26% on MBPP), which directly translates to improved MEC-side accuracy (rising from 81.90 to 88.24% and 71.00 to 73.94%, respectively).

This improvement demonstrates higher routing efficiency, where the router more accurately assesses prompt difficulty and allocates inference tasks between edge and

Table 9 Routing rate and accuracy rate at MEC side in percentage (%)

	Routing rate (%)		Accuracy rate at MEC (%)	
	HumanEval	MBPP	HumanEval	MBPP
No-Aug	74.39	41.80	81.90	71.00
Aug	79.27	50.26	88.24	73.94

cloud models, leading to better utilization of computational resources and higher overall success rates.

5 Discussion and limitations

5.1 Performance comparison

The cascading-based approach provides better program accuracy compared to the routing-based approach and achieves accuracy comparable to the cloud-only approach as the number of evaluated unit tests increases. However, the cascading approach is time-intensive because it involves three stages: prompt inference, unit test generation, and unit test evaluation. Additionally, if the program generated by the weaker LLM is deemed inadequate, the same prompt is passed to the stronger LLM, effectively doubling the runtime required to generate the final program. Despite this, the overall performance of the cascading approach, particularly for higher values of k in unit test evaluations, is similar to that of the cloud-only approach.

Thus, to quantify the Program Generation Efficiency, J, the accuracy/runtime ratio is computed, and tabulated in Table 10.

Based on Table 10, both the No-Aug. Router and Aug. Router approaches demonstrate higher program generation efficiency than the Cascade approach and are competitive with the MEC-Only and Cloud-Only approaches. This indicates that our proposed router-based approach successfully learns the limitations of the weaker LLM and effectively offloads challenging prompts to a stronger LLM, while maintaining competitive routing accuracy.

Table 10 Program generation efficiency, J, for each offloading approach. Higher value indicates higher efficiency

Offloading approach	HumanEval	MBPP
MEC-only	2.37	2.22
Cascade	0.18	0.07
No-Aug. router	3.49	1.32
Aug. router	2.87	1.01
Cloud-only	3.48	0.87

5.2 Methodology justification

The mask-based fill augmentation technique based on GraphCodeBERT [15] was selected primarily for its simplicity, speed of implementation, and transparent mechanism, which make it well-suited for this study. Compared to a more complex augmentation approaches such as GAN-based synthesis, paraphrasing, or rule-based transformations, the mask-fill method can be implemented directly using existing pretrained masked language models without additional adversarial training or complex model tuning [19]. This enables faster experimentation and easier interpretation of how data variability influences router generalization.

In contrast, CodeBERT-Python was chosen for fine-tuning the router since it is pretrained specifically on Python repositories, aligning closely with our dataset and enabling better classification of Python-related prompts. Moreover, CodeBERT-Python has a lighter architecture than GraphCodeBERT, making it more suitable for repeated inference and fine-tuning in edge deployment scenarios. Larger models such as CodeT5 [20] or other models could improve performance but would increase computational cost, which is beyond the scope of this study.

5.3 Limitations

The current runtime and efficiency analysis does not incorporate external factors such as network latency, cloud service cost, or energy consumption, all of which can substantially affect real-world performance in distributed edge–cloud environments. Consequently, the reported runtime results primarily reflect computational efficiency under controlled laboratory conditions.

Additionally, this study focuses exclusively on code generation tasks using the HumanEval and MBPP datasets. This focus stems from the availability of objective, test case—based evaluation metrics that enable consistent labeling of prompt difficulty. Extending the framework to broader NLP tasks—such as question answering or summarization—would require suitable task-specific evaluation criteria (e.g., BLEU or ROUGE scores) to produce equivalent supervision signals.

Finally, while the proposed efficiency metric $J = \text{accuracy}/\text{runtime}$ provides a concise way to compare computational performance, it remains a one-dimensional measure. The ratio combines dimensionless and time-based quantities, which may obscure the inherent Pareto trade-off between accuracy and latency. Moreover, small variations in runtime can disproportionately affect J, and the metric does not capture operational factors such as API wait time, monetary cost, or energy usage. Future work will explore a more comprehensive, multi-objective efficiency formulation to better reflect practical deployment constraints.

6 Conclusion

This paper presents a novel data augmentation strategy to enhance routing-based prompt offloading in large language model (LLM) systems. By directing simpler prompts to smaller, less resource-intensive LLMs and reserving more complex prompts for larger, more capable models, our method effectively balances accuracy and efficiency. This targeted allocation ensures that each prompt is processed by the most appropriate model, leading to optimized program generation performance. Experimental results demonstrate that our approach achieves up to a 16× improvement in efficiency compared to cascaded offloading strategies, while preserving inference accuracy. Overall, the proposed method significantly reduces computational costs without compromising output quality, offering a practical and scalable solution for multi-LLM deployment.

Acknowledgements A part of this paper is based on results obtained from the project, "Research and Development Project of the Enhanced infrastructures for Post-5G Information and Communication Systems" (JPNP20017), commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

Author contributions Muhammad Syafiq Mohd Pozi played significant role in conceptualization, methodology, programming, validation, investigation, writing - original draft, writing - review & editing, and visualization of the main manuscript Yukinori Sato played significant role in funding acquisition, Supervision, Project administration, Conceptualization, Writing — review & editing of the main manuscript.

Funding Open access funding provided by The Ministry of Higher Education Malaysia and Universiti Utara Malaysia.

Data availability No datasets were generated or analyzed during the current study.

Declarations

Conflict of interest The authors declare no Conflict of interest.

References

1. Zhang H, Li X, Bing L (2023) Video-llama: an instruction-tuned audio-visual language model for video understanding. In: Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pp. 543–553
2. Jain N, Vaidyanath S, Iyer A, Natarajan N, Parthasarathy S, Rajamani S, Sharma R (2022) Jigsaw: large language models meet program synthesis. In: Proceedings of the 44th International Conference on Software Engineering, ICSE '22, p. 1219–1231

3. Chen M, Tworek J, Jun H, Yuan Q, de Oliveira Pinto HP, Kaplan J et al (2021) Evaluating large language models trained on code. arXiv:abs/2107.03374
4. Majdinasab V, Bishop MJ, Rasheed S, Moradidakhel A, Tahir A, Khomh F (2024) Assessing the security of github copilot's generated code-a targeted replication study. In: 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 435–444. IEEE
5. Okuda K, Amarasinghe S (2024) Askit: Unified programming interface for programming with large language models. In: Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization, CGO '24, p. 41–54
6. Alliance AR (2024) Ai-ran alliance vision and mission white paper. Techical Report
7. Achiam J, Adler S, Agarwal S, Ahmad L, Akkaya I, Aleman FL, Almeida D, Altenschmidt J, Altman S, Anadkat S et al (2023) Gpt-4 technical report. arXiv:2303.08774
8. Chen L, Zaharia M, Zou J (2023) Frugalgpt: How to use large language models while reducing cost and improving performance. arXiv:2305.05176
9. Liu J, Xia CS, Wang Y, Zhang L (2024) Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. Advances in Neural Information Processing Systems. 36
10. Chen B, Zhu M, Dolan-Gavitt B, Shafique M, Garg S (2024) Model cascading for code: Reducing inference costs with model cascading for llm based code generation. arXiv:2405.15842
11. Pozi MSM, Sato Y (2024) A case for deploying dynamic neural network on edge-cloud continuum environment. In: 2024 IEEE International Conference on Edge Computing and Communications (EDGE), pp. 92–98. IEEE
12. Sakota M, Peyrard M, West R (2024) Fly-swat or cannon? Cost-effective language model choice via meta-modeling. In: Proceedings of the 17th ACM International Conference on Web Search and Data Mining, WSDM '24, p. 606–615. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3616855.3635825
13. Si C, Shi W, Zhao C, Zettlemoyer L, Boyd-Graber J (2023) Getting more out of mixture of language model reasoning experts. arXiv:2305.14628
14. Liu J, Xia CS, Wang Y, Zhang L (2023) Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In: Thirty-seventh Conference on Neural Information Processing Systems. https://openreview.net/forum?id=1qvx610Cu7
15. Guo D, Ren S, Lu S, Feng Z, Tang D, Liu S, Zhou L, Duan N, Svyatkovskiy A, Fu S, Tufano M, Deng SK, Clement C, Drain D, Sundaresan N, Yin J, Jiang D, Zhou M (2021) Graphcodebert: Pretraining code representations with data flow. https://arxiv.org/abs/2009.08366
16. Zhou S, Alon U, Agarwal S, Neubig G (2023) Codebertscore: Evaluating code generation with pretrained models of code. arXiv:2302.05527
17. Liu C, Belkin M (2018) Accelerating sgd with momentum for over-parameterized learning. arXiv:1810.13395
18. Feng Z, Hu Y, Yang X, Zhang S et al (2020) Codebert: A pre-trained model for programming and natural languages. In: Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 1160–1164. ACM
19. Shorten C, Khoshgoftaar TM, Furht B (2021) Text data augmentation for deep learning. J Big Data 8(1):101
20. Wang Y, Wang W, Joty S, Hoi SC (2021) Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv:2109.00859