CoDyn: Dynamic LLM Routing for Coding Tasks

Mirazul Haque [*†]
J.P.Morgan AI Research

Petr Babkin [†]
J.P.Morgan AI Research

Vali Tawosi
J.P.Morgan AI Research

Saba Rahimi
J.P.Morgan AI Research

Natraj Raman
J.P.Morgan AI Research

Xiaomo Liu
J.P.Morgan AI Research

Abstract

Large language models (LLMs) have become integral tools in various business applications, including software engineering, due to their ability to process and generate text. However, the diverse landscape of LLMs, encompassing both proprietary and open-source models with varying architectures and training methodologies, presents opportunities for optimizing cost-performance through model routing. Model routing is a meta-learning paradigm that dynamically selects the optimal model based on user prompts and preferences, leveraging the strengths of different LLMs for specific tasks. Although model routing has gained popularity in natural language tasks, its potential has not been extensively explored for software engineering tasks. In this study, we investigate the dynamic routing capability for various code-based tasks. Initially, we select five models and assess their effectiveness across five coding-related tasks. To create the router, we fine-tune low-cost LLMs using three distinct fine-tuning techniques. We then evaluate these techniques based on three research questions. Our experimental results demonstrate that LLM Classifier-based router consistently match or surpass the effectiveness of the strongest model while offering 43% average cost savings and predictably scaling with varying the cost weight hyperparameter to achieve even greater savings for a moderate degradation in task effectiveness.

1 Introduction

In recent years, large language models (LLMs) have become powerful tools for business tasks involving language, including software engineering. In typical enterprise setups or AI code editors, user prompts are usually sent to the single best model available. However, the diverse landscape of proprietary and open-source models of various sizes, architectures, and training recipes offers opportunities for cost-performance optimization. Model routing has emerged as a meta-learning paradigm aimed at dynamically selecting the optimal model based on user prompts and preferences [Chen et al., 2023], [Hari and Thomson, 2023], [Ong et al., 2024]. Routing is based on insights that: a) different LLMs excel at different tasks due to their training, often outperforming the best overall model; and b) within the same task, prompts vary in difficulty, allowing smaller and cheaper models to match the performance of the strongest model on a carefully selected subset.

Routing mechanisms can be broadly divided into two types based on their working mechanism. The first type, non-predictive routing, evaluates the output of weaker models and uses stronger models if the output quality does not meet a specific threshold [Jiang et al., 2023a]. The second type, predictive routing, aims to balance performance with cost by employing predictive techniques to infer each model's expected response quality based solely on the query [Shnitzer et al., 2023], [Hari and Thomson, 2023], [Ding et al., 2024]. Predictive routing is more challenging to implement because the router prediction is based only on the query, but it is also more cost-effective. Implementations of

[*]Email of corresponding author mirazul.haque@jpmchase.com
[†]These authors contributed equally to this work.

routing mechanisms range from training-free similarity-based scoring [Zhao et al., 2024], [Ong et al., 2024] and simple classifiers [Shnitzer et al., 2023], to LM classifiers and Autoregressive LLMs [Ong et al., 2024].

Despite a large body of literature on LLM routing, the majority of work is limited in scope to the general NLP domain, e.g., benchmarks such as MMLU [Hendrycks et al., 2020], HELM [Liang et al., 2022], COQA [Reddy et al., 2019], or corpora The Pile [Gao et al., 2020], with scarcely any attention to code-specific evaluation. Most of these evaluations are done vs discrete labels, MLM, and even MixInstruct comprising datasets like Dolly containing coding questions, are limited in evaluation to text-based similarity via BERT/BART-score.

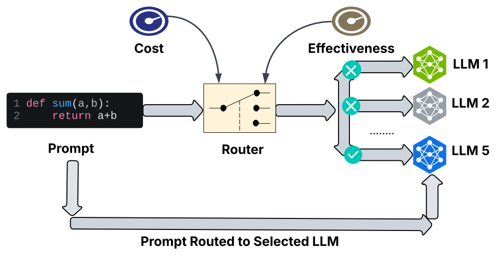

```
def sum(a,b):
    return a+b
```

Figure 1: Overview of Dynamic Routing

Whereas code quality is known to poorly correlate with naive lexical overlap-based or coarse semantic similarity metrics, instead necessitating the use of specialized syntactically-aware (CodeBLEU) and execution-based (pass@k) metrics. Additionally, most routing problem formulations [Ong et al., 2024] assume a binary decision between exactly one strong and one weak model, whereas allocating among n progressively larger/more costly models could offer a more balanced cost-performance tradeoff and robust routing 1. To address this gap in literature, in this work, we focus on exploring the effect of predictive dynamic routing mechanisms on different coding tasks and five target LLMs.

First, we select five Llama [Grattafiori et al., 2024] LLMs with different parameter size as target LLMs: 3-1-8B, 3-2-11B, 3-70B, 3-1-70B, 3-1-405B. Next, we benchmark these models against five coding related tasks. Then we analyze the diversity of the task effectiveness for these tasks and train cost-effective LLMs on these five datasets with different finetuning mechanisms and finally discuss the results.

We evaluate different finetuning techniques based on three research questions. First, how proficient is the router in accurately identifying the most effective large language models (LLMs)? Second, To what extent can efficiency be enhanced by utilizing the router? Third, we discuss the extent of diversity in model selection by the router in dynamic routing (in Appendix). To evaluate these three research questions, we finetune the routers with different cost settings.

Here are the contributions of our work.

- To the best of our knowledge, this is the first work that extensively evaluates dynamic routing capability in coding tasks.
- We have designed novel reward functions that can be used to modify auto-regressive models for model routing.
- We evaluate the dynamic routing capability of three different techniques across different cost settings.

The rest of the work is organized as follows. In Section 2, we discuss the related works in the field of dynamic routing. In Section 3, we explain about all the datasets that have been considered for routing tasks. In Section 4, we explore a preliminary study on the discussed datasets based on performance of different models on these datasets. After finalizing the datasets, we discuss different methodolgies for router training Section 5. Finally, we evalaute different routing techniques in Section 6.

2 Related Work

In the realm of model selection, two primary approaches have emerged: predictive and non-predictive strategies. Predictive methods, as discussed in works such as [Shnitzer et al., 2023], [Hari and Thomson, 2023], and [Ding et al., 2024], focus on selecting the optimal large language model (LLM) without the need to evaluate the output explicitly. These methods rely on pre-trained criteria or heuristics to make decisions, thereby streamlining the selection process. In contrast, non-predictive approaches, highlighted in studies like [Chen et al., 2023], [Aggarwal et al., 2023], and [Yue et al., 2023], assess the quality of responses from an initial model to determine if more advanced models are necessary. Despite their differences, a commonality across these methodologies is the necessity to train the router LLM, which requires a robust training dataset. In this context, MixInstruct has been employed as a dataset, as noted in [Jiang et al., 2023a], providing a comprehensive foundation for training and enhancing the router's decision-making capabilities.

However, all these related works do not concentrate on coding related tasks. As code related tasks can be of many types and can be more complex to analyze than natural language tasks, it becomes very important to explore the capability of dynamic routing tasks for code related tasks.

3 Dataset Creation

Creation of a dataset for routing, revolves around defining the following three key ingredients: a) datasets to source diverse tasks from, b) base models used for generating responses of varying quality and cost, and c) metrics used to derive the preference over the models' responses.

3.1 Code Tasks

Coding tasks can be distinguished based on a large number of characteristics, including the programming language used, code size and complexity, as well as the application domain. We chose to optimize for the diversity in the semantics of the task itself, as that will help us uncover qualitatively diverse capabilities of code LLMs. Specifically, code generation focuses on producing code that performs a computation in accordance with user's instructions. By contrast, the goal of automatic program repair is to identify and rectify bugs causing the code to produce the wrong result. On the other hand, vulnerability detection challenges the model to understand subtle weaknesses of the code to make a determination if it is vulnerable to malicious exploits, even if it computes the correct result. For each of these tasks we use the following datasets.

APPS [Hendrycks et al., 2021], a code generation dataset of Python solutions to 10k problems, spanning introductory coding, interviews and code competitions. Each problem is accompanied with a detailed problem description and sample inputs/outputs, and correctness is checked against a test suite. We selected 5k examples for analysis from APPS.

RunBugRun [Prenner and Robbes, 2023] is a program repair dataset, derived from CodeNet [Puri et al., 2021] code competitions, comprising nearly a quarter of million submissions in 8 programming languages, encompassing 4000 distinct problems. Each problem is accompanied with an extensive inventory of unit tests, which we leverage to judge correctness of LLM responses. In this work we use 2k bugs from the Python validation split.

SVEN [He and Vechev, 2023], a curated dataset of 803 vulnerable/fixed program pairs in C/C++ or Python exhibiting one of 9 critical vulnerabilities identified in the MITRE-25 list, each having at least 40 fixes, used for training a security-aware controlled generation system. We use 322 python based program pairs for analysis.

From the above three tasks, APPS and RunBugRun are Code-generative tasks, and SVEN is a discriminative task. To ensure diversity of our dataset we imported two additional tasks from CodeXGLUE meta-dataset [Lu et al., 2021], for code-generative tasks: Java code refinement and Java-to-C# function translation. From each of these additional datasets, we aimed at sampling 5k initial examples from the test split, whenever available.

3.2 Models

We consider five LLama models for our experimentation: Llama-3.1-8B, Llama-3.2-11B, Llama-3.1-70B, Llama-3-70B, and Llama-3.1-405B [Grattafiori et al., 2024]. The models' parameter counts range from 8B to 405B, which gives us a wide range to experiment with in terms of optimizing quality and cost. The reason behind selecting these models is that all the models are from the same open source LLM family and their availability in Amazon Bedrock. As one of the main criteria behind our ranking metric is cost, these models' availability in Amazon Bedrock ensures that we have a very specific per-token cost for each model usage. In order to obtain preference data, we prompt each of these models on every example from our dataset and analyze their responses based on the metrics defined in the next section.

3.3 Performance Metrics

In order to train a router model, we need preference data reflecting the effectiveness of each of five models on each of the input prompts, which will enable us to rank them in the order of preference. What specific effectiveness measure is appropriate depends on the type of output expected for each task (e.g., code, natural language, or a categorical label).

Code-Generative. When it comes to evaluating generated code, the gold standard is to report success rate derived from running the candidate solution through a set of unit tests. Multiple ways of aggregating these results exist in the literature, including reporting a fraction of passing tests [Hu et al., 2020], fraction of programs passing all tests [Jiang et al., 2023b], or the probability of sampling a passing solution among multiple candidates [Kulal et al., 2019]. We use the fraction of generated programs that pass all test cases as the quality metric for the Run Bug Run and APPS datasets.

When unit tests are not available, generated code needs to be compared against a reference solution. Evaluators implemented as part of CodeXGLUE largely utilize either exact textual match or BLEU score for generative tasks, which is known to be suboptimal for evaluating code correctness. For this reason, we resort to using the more code-appropriate metric CodeBLEU, which is a weighted average of n-gram overlap, as well as syntactic and data flow match [Ren et al., 2020] on code refinement and translation.

Categorical. When the expected output for the task is a categorical label, as in vulnerability detection, we can instruct the LLM to output a special token such as True/False, and postprocess its response to extract a binary value. Then we can apply standard classification metrics such as accuracy to measure response quality.

4 Preliminary Study

To ensure feasibility of effective model routing, we conduct a preliminary study assessing the diversity of task effectiveness scores across chosen model sizes. We consider five datasets—APPS, CODE REFINEMENT, CODE TRANSLATION, RUNBUGRUN, SVEN—each providing per-instance scores for a fixed portfolio of models. Let s_{ij} denote the score of model j on instance i.

SBS, Oracle, and Oracle Gap. Following the algorithm-selection framework [Kerschke et al., 2019], we summarize two canonical baselines per dataset: (i) the *Single Best Solver* (SBS), i.e., the single model with the largest mean score; and (ii) the *Oracle*, i.e., the per-instance oracle that selects $\max_j s_{ij}$ on each instance. The *oracle gap* (OG) measures the headroom available to any router:

$$\text{OG} \;=\; \frac{1}{n}\sum_{i=1}^{n}\max_{j} s_{ij} \;-\; \max_{j}\frac{1}{n}\sum_{i=1}^{n} s_{ij}, \tag{1}$$

where n is the number of instances. Intuitively, OG > 0 indicates potential gains from per-instance routing; the larger the gap, the larger the achievable improvement upper bound. Before training any router, we compute OG for all datasets. We also report a diagnostic *winner-change rate* (WCR): the fraction of instances where the per-instance best model differs from the SBS model. High WCR confirms that instance-dependent choices actually occur substantially.

	APPS	TRANS	REFI	RBUGR	SVEN
SBS mean	0.1642	0.0680	0.1826	0.3777	0.5057
Oracle mean	0.5101	0.3689	0.6803	0.5866	1.0608
OG (abs)	0.3460	0.3009	0.4977	0.2089	0.5551
WCR	0.898	0.868	0.750	0.785	0.376

Table 1: **Oracle-gap screening.** SBS = best single model by mean; Oracle = per-instance oracle; OG = Oracle-SBS (absolute units). WCR = fraction of instances where the per-instance winner $\neq$ SBS. All five datasets exhibit non-negligible OG, indicating clear headroom for routing.

Results on the five datasets. Table 1 summarizes SBS, VBS, absolute oracle gap (OG), and WCR for the five datasets. All five datasets show substantial headroom, indicating that per-instance routing is well-justified prior to training any router.

5 Methodology

When it comes to devising a router model, multiple architectures have been suggested in the literature, including ones based on low-cost LLMs [Ong et al., 2024] — which is an appealing option for prompt-conditioned routing. Drawing inspiration from the RouteLLM framework, we focus on two implementation approaches: an LLM classifier/regressor (CLS, henceforth), which outputs either a label or score given a predefined set of models; and an autoregressive decoder, which generates tokens that correspond to model identifiers. For the latter approach we further explored two training techniques: supervised finetuning (SFT), and group relative policy optimization (GRPO) [Guo et al., 2025]. Figure 6 (in appendix) schematically depicts all three approaches.

5.1 LLM Classifier

The goal of routing is to optimally allocate user queries among models to balance response quality, cost, or other criteria [Hari and Thomson, 2023]. Traditional routing assumes a single weak and strong model, with the router interpolating between them [Ong et al., 2024]. Instead, we define routing as an n-way ranking problem with models $m_1, m_2, \ldots, m_n$. Each model receives a score for a given prompt, and the classifier predicts these scores, selecting the model with the highest score. We use an LLM with n regression heads and mean square error (MSE) for fine-tuning. The loss for each regression head is: $\mathcal{L}_i = \frac{1}{N}\sum_{k=1}^{N}\left(\hat{S}_i(x_k) - S_i(x_k)\right)^2$, where $\mathcal{L}_i$ is the MSE loss for the i-th regression head, N is the number of samples, x_k is the k-th prompt, $\hat{S}_i(x_k)$ is the predicted score, and $S_i(x_k)$ is the true score for model m_i. The overall loss for the classifier is the sum of the MSE losses across all regression heads: $\mathcal{L}_{\text{total}} = \sum_{i=1}^{n}\mathcal{L}_i$.

Our configuration seeks to predict (and rank models based on) only the expected score. The expected score can be only based on model effectiveness or can be a mixture of effectiveness and cost. In the Section 6, we discuss different settings for calculating scores. For training the LLM Classifier, we use full-weight finetuning, *i.e.,* all the weights in the model can be modified during finetuning. We select this configuration because earlier NLP classifier models like BERT [Devlin et al., 2019] is finetuned using full-weight finetuning.

5.2 Autogressive Model

In autoregressive decoding, we fine-tune the LLM to output a sequence of model identifiers in order of preference in response to a prompt. Unlike the LLM Classifier, where full model weights are used for fine-tuning, we employ Low-Rank Adaptation (LoRA) for fine-tuning autoregressive models. While the LLM Classifier uses a fixed number of regression heads, the autoregressive model's output tokens depend on previous tokens, increasing memory costs. Therefore, different adaptation methods are used for fine-tuning autoregressive models.

Low-Rank Adaptation (LoRA) is an innovative approach that enhances the efficiency and adaptability of autoregressive models. It reduces computational and memory overhead by introducing low-rank matrices into the model's architecture, allowing efficient adaptation of pre-trained models to specific tasks without updating all model parameters. This significantly reduces resource requirements. The low-rank matrices introduced by LoRA act as a compact and efficient means of capturing task-specific information, enabling the model to fine-tune its outputs with minimal computational cost. With LoRA adaptation, we fine-tune the router model using two mechanisms: Supervised Fine-Tuning (SFT) and Group Relative Policy Optimization (GRPO).

5.2.1 Supervised Fine-Tuning (SFT)

SFT is one of the foundational steps in aligning a language model with task-specific behavior. Given a pretrained model f_θ, parameterized by θ, SFT adapts it using a labeled dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$, where each x_i is an input prompt and y_i is its corresponding target output. The objective is to minimize the cross-entropy loss: $\mathcal{L}_{\text{SFT}}(\theta) = -\sum_{i=1}^{N} \log p_\theta(y_i \mid x_i)$. This encourages the model to mimic high-quality demonstrations. However, if the dataset lacks coverage or contextual diversity, the model may not generalize well to new or nuanced examples.

5.2.2 Group Relative Policy Optimization (GRPO)

GRPO extends SFT by learning from preferences between outputs instead of relying solely on fixed targets. For a given prompt x, assume the model produces two candidate outputs: y^+, which is preferred, and y^-, which is less preferred. A reward function $r_\theta(x, y)$ evaluates the quality of a response. GRPO minimizes the following preference-based loss:

$$\mathcal{L}_{\text{GRPO}}(\theta) = -\log \left(\frac{\exp(r_\theta(x, y^+))}{\exp(r_\theta(x, y^+)) + \exp(r_\theta(x, y^-))} \right).$$

This formulation encourages the model to produce responses that align better with user-defined or task-specific criteria, offering more flexible and targeted alignment than SFT alone.

For GRPO reward function, we have used two different reward functions. In the algorithm 1 (In appendix), we show the reward functions. In the first reward function, we ensure the output format of the router is correct, *i.e.,* the models are separated by comma. In the next reward function, we try to ensure that the router predicts one model within top three models based on ranking scores.

6 Evaluation

We evaluate the dynamic routing capabilities of various models by addressing the following research questions:

RQ1. How effective is the router in predicting which large language model (LLM) will be effective on a given prompt?

RQ2. To what extent can efficiency be enhanced by utilizing the router?

Additionally, we explore the following third research question and added the section in appendix A.3.

RQ3. What is the extent of diversity in model selection by the router in dynamic routing, and does it consistently choose different models across various scenarios?

6.1 Setup

6.1.1 Models and Baselines

To optimize our dynamic routing system, we employ various cost-effective models as routers. For evaluating the three training strategies mentioned earlier, we utilize the Qwen family of models [Yang et al., 2024], specifically the Qwen-2.5-8B and Qwen-2.5-1.5B models, across all router configurations, including the LLM Classifier. This ensures a fair comparison in terms of the number of parameters.

For LLM Classifier, we add five dedicated regression heads (one for each LLama model) on top of the penultimate transformer layer to replace the standard LM-heads used for next token prediction. In the Supervised Fine-Tuning (SFT) and Gradient-based Policy Optimization (GRPO) techniques, these models are used as autoregressive decoders and are fine-tuned to enhance their performance.

Traditional Models. As a baseline for traditional model architecture, we used the RoRF library [NotDiamond]. RoRF is a Random Forest classifier designed to analyze evaluation data from large language models (LLMs) and learn a mapping from prompt embeddings to the most suitable model for a given prompt. The classifier predicts the likelihood of one of four possible outcomes for a pair of models: **Label 0:** Model A is correct, while Model B is incorrect. **Label 1:** Both Model A and Model B are incorrect. **Label 2:** Both Model A and Model B are correct. **Label 3:** Model A is incorrect, while Model B is correct. RoRF class probabilities are estimated empirically by computing the proportion of decision trees (estimators) that predict each label.

As this is a binary predictor, we evaluate this technique with two settings. It is not possible to directly compare it to our 5-model setup, so we report results for two realistic scenarios commonly used in literature. In the first configuration, we provide scores for the 3-1-8B and 3-1-405B models, reflecting

Table 2: **Effectiveness and cost values of all evaluated techniques on all datasets. Effectiveness is represented by E (higher is better) and cost is represented by C (lower is better). No Cost, 0.25 Cost, 0.5 Cost stand for Setting 1, Setting 2, and Setting 3 respectively.**

Technique	Setting	SVEN(E)↑	SVEN(C)↓	RBR(E)↑	RBR(C)↓	APPS(E)↑	APPS(C)↓	Trans(E)↑	Trans(C)↓	Refi(E)↑	Refi(C)↓
	No Cost	**0.66**	0.95	0.75	0.84	**0.22**	0.43	**0.38**	0.30	**0.37**	0.33
CLS 0.5	0.5 Cost	0.52	0.10	0.67	0.09	0.20	0.12	0.37	0.06	**0.37**	0.07
	0.25 Cost	0.65	0.77	0.73	0.49	0.20	0.19	**0.38**	0.11	**0.37**	0.10
	No Cost	0.64	0.91	**0.77**	0.98	0.21	0.46	0.37	0.49	**0.37**	0.47
CLS 1.5	0.50 Cost	0.51	0.02	0.67	0.08	0.18	0.08	**0.38**	0.06	**0.37**	0.07
	0.25 Cost	0.56	0.30	0.74	0.78	0.17	0.14	**0.38**	0.05	**0.37**	0.08
	No Cost	0.04	0.12	0.06	0.15	0.04	0.08	0.01	0.50	0.01	0.14
SFT 0.5	0.50 Cost	0.02	0.2	0.04	0.25	0.04	0.09	0.01	1	0.01	0.52
	0.25 Cost	0.03	0.04	0.04	0.10	0.03	0.11	0.01	0.34	0.01	0.51
	No Cost	0.05	0.17	0.07	0.27	0.04	0.07	0.01	0.52	0.01	0.13
SFT 1.5	0.5 Cost	0.01	0.2	0.04	0.19	0.04	0.09	0.01	1	0.02	0.36
	0.25 Cost	0.03	0.02	0.05	0.22	0.03	0.08	0.003	0.36	0.01	0.53
	No Cost	0.10	0.51	0.66	0.2	0.18	0.13	0.34	0.49	**0.37**	0.88
GRPO 0.5	0.5 Cost	0.05	1	0.42	0.54	0.13	0.35	0.36	1	**0.37**	0.92
	0.25 Cost	0	0	0.63	0.55	0.15	0.17	0.34	1	**0.37**	0.99
	No Cost	0.37	0.91	0.67	0.22	0.19	0.14	0.34	0.82	**0.37**	0.94
GRPO 1.5	0.5 Cost	0.53	0.96	0.66	0.15	0.2	0.1	0.34	0.93	**0.37**	0.86
	0.25 Cost	0.32	0.28	0.68	0.21	0.19	0.1	0.34	0.73	**0.37**	0.97
Random		0.41	0.26	0.59	0.26	0.15	0.267	0.34	0.26	0.36	0.29
Oracle		0.91	0.64	0.85	0.13	0.35	0.09	0.40	0.06	0.41	0.09
RoRF (8b-405b)		0.51	0.89	0.54	0.07	0.12	0.013	**0.38**	0.70	**0.37**	0.89
RoRF(70b-405b)		0.50	0.89	0.55	0.26	**0.22**	0.22	0.36	0.25	0.36	0.42

the strongest vs. weakest setting. In the second configuration, we provide scores for the 3-1-70B and 3-1-405B models, reflecting the strongest vs. runner-up scenario.

6.1.2 Cost Metric

To evaluate our routing approach, we consider one metric corresponding to each research question. We assess the overall effectiveness by aggregating scores of router-predicted models for each prompt. Reporting intrinsic metrics like top-1 accuracy for each router architecture is possible, but may not provide valuable insights without considering the actual scores achieved by each model on a task.

For evaluating the routing process's overall cost, literature commonly reports the percentage of calls to the strongest (most expensive) model. In our multi-model setting, we calculate the weighted average of each model's cost by the fraction of calls routed to it, normalized by the cost of the most expensive model:

$$\frac{\sum_{i=1}^{n} (\text{\% of calls to model}_i \times \text{model\_cost}_i)}{\text{highest model\_cost}} \tag{2}$$

Defining model cost is complex and involves multiple factors. In our experiments, we considered model costs based on the price per million output tokens in Amazon Bedrock (as of December 2024). Using this value, we calculated the cost ratio for models 3-1-8B, 3-2-11B, 3-1-70B, 3-70B, and 3-1-405B as 1, 1.6, 5, 16, and 72, respectively.

6.1.3 Cost Settings

Defining the routing problem as ranking based on preference scores allows for customization. In the simplest case, preference is defined by effectiveness score alone, ignoring cost. In cost-aware settings, preference is a weighted combination of cost and effectiveness. Since higher effectiveness and lower cost are preferred, cost is replaced with a quantity like affordability, denoted as $CS^{-1} = 1 - \frac{CS-1}{max(CS)-1}$, which ranges from 0 to 1. The ranking score for routing is: $RS = \lambda_1 * ES + \lambda_2 * CS^{-1}$

Hence the cost-free setting can be thought of as fixing λ_1 to 1, while setting λ_2 to 0 (Setting 1). While it could be tempting to incorporate cost into our preference, by setting its weight equal to that of effectiveness, it is easy to see this would make a completely wrong output by the cheapest model be equally preferable as a perfect output by the most expensive one. Hence, we experiment with two weight settings favoring effectiveness: $\lambda_2 = 0.25$ (Setting 2) and $\lambda_2 = 0.5$ (Setting 3).

6.1.4 Dataset

As discussed in Section 4, we selected five datasets for fine-tuning and evaluating different techniques: SVEN, APPS, RunBugRun, Code Refinement, and Code Translation. Although we initially started with a higher number of samples for each dataset, the collected data points have been reduced due to failed executions. A detailed description can be found on appendix.

6.1.5 Training

For training purpose, we have used a single 24GB A10G GPU. We finetune our regressor from pretrained checkpoints of Qwen2.5 with the standard MSE loss for three epochs and with a linear warmup for 500 steps, in 16bit precision using maximum batch that fits on a single 24GB A10G GPU (4 for 0.5B and 1 for 1.5B models). For SFT Training, we use cross entropy loss and train the model for three epochs. Also we use maximum batch size of 4 for both models. For GRPO, we run the model for two epochs. For each step, we generate four outputs to compare based on the reward. This makes the training time of GRPO significantly higher than the other two methods. For all three training techniques, we have used AdamW optimizer with the learning rate of $2e-5$. We have fixed the input token length with 2048.

6.2 RQ1. Effectiveness

To evaluate our routing approaches, we report mean effectiveness score of each router configuration on different coding tasks. In addition to comparing the three LLM finetuning techniques, we report the traditional machine learning based technique RoRF, random routing and Oracle. In Oracle, we consider a router having access to gold preference data that would always predict the model with highest effectiveness and lowest cost score, for any given prompt. This establishes the upper bound of scores in principle achievable on each dataset. Notably, Oracle effectiveness always exceeds that of the single strongest model (Llama3.1-405B), however Oracle cost can still be higher than that of cheaper models, indicating a portion of prompts require larger models to achieve the maximum score. We also note that for some tasks like Run Bug Run and SVEN, the Oracle sets a high ceiling of performance, while of others like APPS, code refinement, and translation it is considerably lower, suggesting the capabilities of the base LLM to be a limiting factor.

We have reported our detailed findings in Table 2. Additionally, to help highlight the cost-performance trade offs of the best-performing configurations for each approach, we visualize the results on each of datasets in Figures 2–3. With respect to effectiveness, LLM classifier finetuned with Qwen-2.5-0.5B in the cost-free setting (CLS_0), while well below the Oracle level, generally matches or outperforms the strongest model and consistently outperforms all other techniques, including autoregressive decoding (GRPO), as well as RoRF and random baselines. LLM classifier based on the larger Qwen-2.5-1.5B performs similarly to the 0.5B variant, not showing a clear edge due to its increased size. We find that LLM AR model fine-tuned with SFT technique performs worst among the techniques compared. The effectiveness is significantly low across the datasets. We find that the router finetuned with SFT can not complete the routing task properly, *i.e.,* can not give us predicted model. Because of this reason, the technique effectiveness score of 0 for most of the prompts.

Although the GRPO technique also tries to finetune LLM AR model, we find better effectiveness from the routers finetune with GRPO. One of the main reasons behind this is we had designed specific reward functions that can ensure both correct output format and correct model prediction. However, the effectiveness of these models is lower than LLM Classifier models. Specially we find that for SVEN dataset, the effectiveness of the AR models finetuned with Qwen-2.5-0.5B is significantly lower. Also, for Setting 2 and Setting 3, we can't find the pattern of lower effectiveness in these router models.

For RoRF technique, we can find similar effectiveness scores for both configurations. While the technique is effective on selecting models for Code Refinement and Code Translation, the overall effectiveness of this technique is limited. Also, for understanding coding tasks, the effectiveness of traditional models like random forest can be limited.

RQ1 Summary. 0.5B LLM Classifier outperforms other routing techniques, consistently matching or surpassing the strongest single model.

6.3 RQ2. Efficiency

We measure the cost efficiency of the different techniques in accordance with Eq. 2, mentioned in Subsection 6.1, relative to the most expensive model. Similarly to RQ1, the cost measurement has been reported in Table 2 and visualized for select configurations in Figures 2–3.

In the cost-free setting, the 0.5B LLM Classifier's cost averages 43% of the largest model's cost, which still represents considerable savings. By increasing the weight of the cost component, we observe drastic reductions in cost – matching the Oracle level, at the expense of moderate reduction in effectiveness, still on par with autoregressive decoding and RoRF approaches. The larger 1.5B LLM Classifier exhibits similarl cost scaling with varying cost settings.

For the SFT technique, we can notice lower cost, but because of the poor effectiveness, cost savings from this technique are not meaningful. For the GRPO technique in setting 1, we find generally lower routing cost on Run Bug Run and APPS, and higher on code refinement and translation. When it comes to cost-aware settings 2 and 3, GRPO-trained router does not exhibit predictable cost savings, in fact for the 0.5B model the cost is nearly doubled in four out of five datasets. This could be a side effect of partial rewards assigned for the target model to be ranked at 2nd and 3rd positions, as indicated in Algorithm 1. For both configuration of RoRF, we find that for APPS and RBR datasets the cost of the first configuration is significantly low. As the first configuration is between the lowest cost and highest cost models, it can be assumed that maximum number of times 8B is picked for these two datasets.

Additionally, we find that for three datasets: SVEN, Code Transformation and Code Refinement, router costs are comparatively high on average.

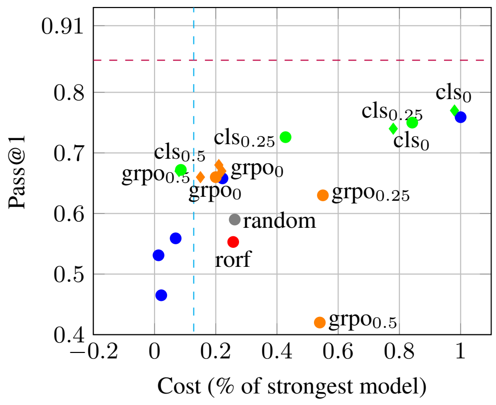


Figure 2: Cost-performance trade-off of different routing approaches on RunBugRun. Unlabeled blue circles represent the 5 Llama3 LLMs of different sizes, while labeled circles correspond to each routing approach. Diamond markers denote 1.5B base model for the router. Subscripts indicate cost weight used. Vertical and horizontal dashed lines represents oracle's cost and performance, respectively.

RQ2 Summary. We can tune the cost of model with different training strategies using LLM Classifier router with average savings of 43% and predictably scaling down with increasing cost weight.

7 Conclusion

This work focuses on optimizing the cost-performance trade-off of LLMs for diverse coding tasks using predictive routing. We create a finetuning-size preference dataset from code generative and discriminative tasks sourced from standalone datasets and CodeXGLUE. Preference scores are generated by running tasks on five Llama3 family LLMs, evaluating responses with execution-based, reference-based, and classification metrics. For a subset of tasks, we train router models using three finetuning approaches: LLM-based score regression, autoregressive decoding with supervised finetuning, and GRPO, combined with two base model sizes and three cost sensitivity settings. Our best routing configurations generally outperform baselines, with preference-regression routing matching or exceeding the strongest single model, offering 43% average cost savings and greater savings with varied cost settings for a moderate effectiveness hit. Further analysis shows our models make diverse routing choices rather than biasing towards a particular model.

8 Acknowledgements

We thank Zhiqiang Ma for constructive feedback.

References

Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, et al. Automix: Automatically mixing language models. *arXiv preprint arXiv:2310.12963*, 2023.

Lingjiao Chen, Matei Zaharia, and James Zou. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.

Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Ruhle, Laks VS Lakshmanan, and Ahmed Hassan Awadallah. Hybrid llm: Cost-efficient and quality-aware query routing. *arXiv preprint arXiv:2404.14618*, 2024.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Surya Narayanan Hari and Matt Thomson. Tryage: Real-time, intelligent routing of user prompts to large language model. *arXiv preprint arXiv:2308.11601*, 2023.

Jingxuan He and Martin Vechev. Large language models for code: Security hardening and adversarial testing. In *ACM CCS*, 2023. URL `https://arxiv.org/abs/2302.05319`.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. Re-factoring based program repair applied to programming assignments. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 388–398. IEEE Press, 2020. ISBN 9781728125084. doi: 10.1109/ASE.2019.00044. URL `https://doi.org/10.1109/ASE.2019.00044`.

Dongfu Jiang, Xiang Ren, and Bill Yuchen Lin. Llm-blender: Ensembling large language models with pairwise ranking and generative fusion. *arXiv preprint arXiv:2306.02561*, 2023a.

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. Impact of code language models on automated program repair, 2023b.

Pascal Kerschke, Holger H Hoos, Frank Neumann, and Heike Trautmann. Automated algorithm selection: Survey and perspectives. *Evolutionary computation*, 27(1):3–45, 2019.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32, 2019.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*, 2022.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

Not-Diamond. Rorf. `https://github.com/Not-Diamond/RoRF`. Accessed: 2025-05-31.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E Gonzalez, M Waleed Kadous, and Ion Stoica. Routellm: Learning to route llms from preference data. In *The Thirteenth International Conference on Learning Representations*, 2024.

Julian Aron Prenner and Romain Robbes. Runbugrun – an executable dataset for automated program repair, 2023.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 2021.

Siva Reddy, Danqi Chen, and Christopher D Manning. Coqa: A conversational question answering challenge. *Transactions of the Association for Computational Linguistics*, 7:249–266, 2019.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. URL `https://arxiv.org/abs/2308.12950`.

Tal Shnitzer, Anthony Ou, Mírian Silva, Kate Soule, Yuekai Sun, Justin Solomon, Neil Thompson, and Mikhail Yurochkin. Large language model routing with benchmark datasets. *arXiv preprint arXiv:2309.15789*, 2023.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

Murong Yue, Jie Zhao, Min Zhang, Liang Du, and Ziyu Yao. Large language model cascades with mixture of thoughts representations for cost-efficient reasoning. *arXiv preprint arXiv:2310.03094*, 2023.

Zesen Zhao, Shuowei Jin, and Z Morley Mao. Eagle: Efficient training-free router for multi-llm inference. *arXiv preprint arXiv:2409.15518*, 2024.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, Binyuan Hui, Niklas Muennighoff, David Lo, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions, 2025. URL `https://arxiv.org/abs/2406.15877`.

A Appendix

A.1 Threats to Validity

We recognize that our results may not generalize to all software development uses of LLMs. However, to mitigate the external validity of the study we, have included five different software engineering tasks in our experiments, all representative of common tasks that LLMs are frequently used for in software development projects. The datasets we used in this study come from popular archives used in several software engineering studies. Also, we conducted an evaluation of several Code to Text examples using different LLMs. Upon manual assessment, we found that the output quality was consistently similar across all five LLMs.

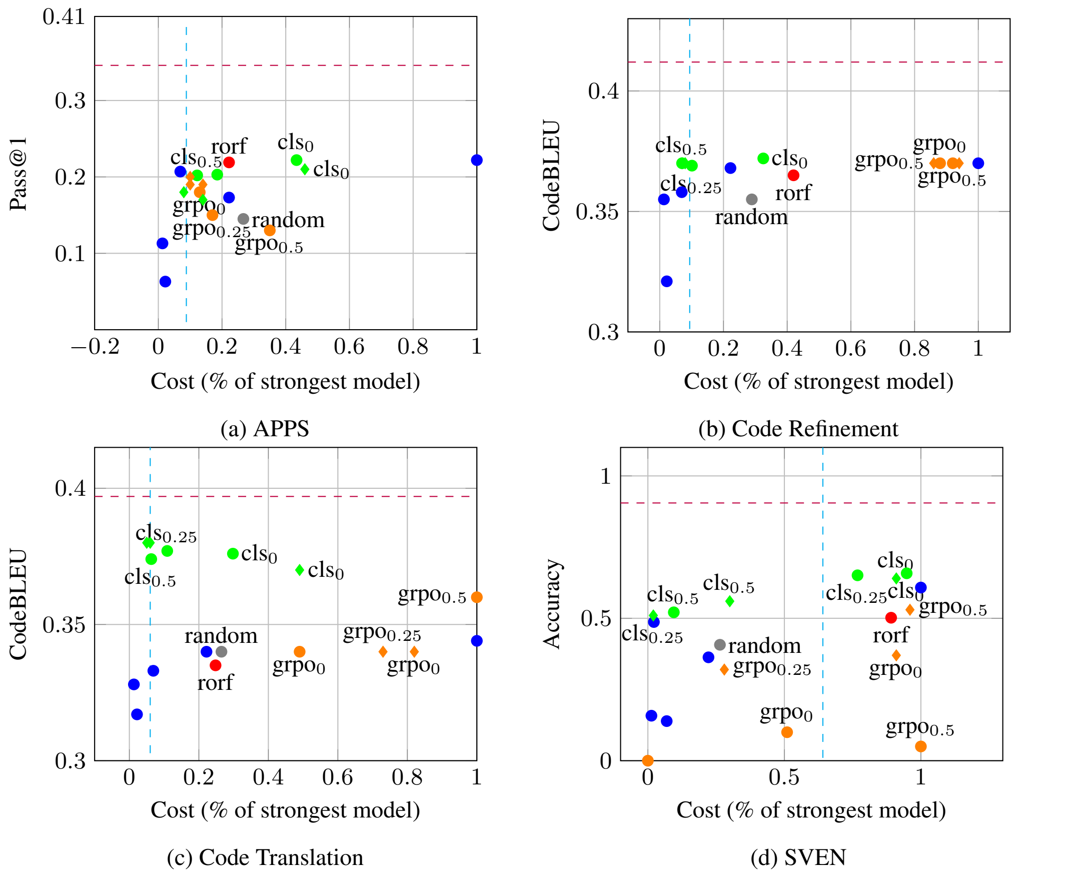
Another valid limitation to this study is the generalizability of the router models to task types not presented in the training data. This is a common limitation for router models, but since we have experimented with multiple task types we argue that retraining the model with new data should perform similarly, although this needs future experiments to validate. We also based our cost model fixed-per-model for a relative order. One might argue that different models may produce different number of output tokens, thus, the actual cost might be different. Note that this information is not known prior to running the new task through the router model, but it could be estimated based on the previous behavior of each model. It is also possible to include such fine-grained cost information in training the router model and evaluate the outcome in the future work.

We haven't added router cost in the total-cost calculation, which might be considered as threat to validity. But, router models used are orders of magnitude smaller than the smallest of the evaluated standalone LLMs and can be deployed on low spec AWS instances or even locally. If we deploy a 0.5B model in EC2 g5.2x server, the system would process approximately 400 tokens/s. Given the input length is 2048 tokens, per prompt would take 5 seconds to process. If we consider the cost of renting g5.2x server per hour, the cost per prompt would be 0.0016 USD, which is significantly less than making the LLM calls. Hence, this cost is not added for calculation.

All our LLMs to route are coming form the same family (i.e., Llama) which may introduce a threat to external validity. In this study, this choice helps us to examine a spectrum of model sizes while fixing the model family, which allows for a more nuanced analysis of the cost-performance tradeoff while minimizes confounding effects of due to model family differences. On a more practical level, model routing requires inference from the selected models, which is later used for training model router. Llama license (specially 3.1) is friendly for using derivative data, while licenses of other models like recent GPT models and Gemini models are not clear about it. Hence, we chose to use Llama family of models.

To mitigate threats to internal validity of the study, we carefully selected appropriate metrics to measure the effectiveness of different LLMs on each task type. These metrics are relevant and valid to measure the quality of response for each task and at the same time reliably distinguish high performing models from low performing ones, which is crucial in training an effective router model.

A.2 Cost-Performance Trade-off For Each Dataset



(a) APPS

(b) Code Refinement

(c) Code Translation

(d) SVEN

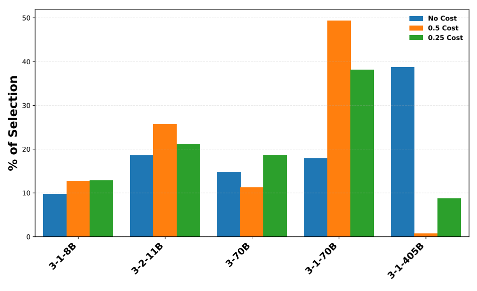
Figure 3: Cost-performance trade-off of various models and routing approaches on generative tasks. Unlabeled blue circles represent the 5 Llama3 LLMs of different sizes, while labeled circles correspond to each routing approach. Diamond markers denote 1.5B base model for the router. Subscripts indicate cost weight used. Vertical and horizontal dashed lines represents oracle's cost and performance, respectively.

A.3 RQ3. Diversity

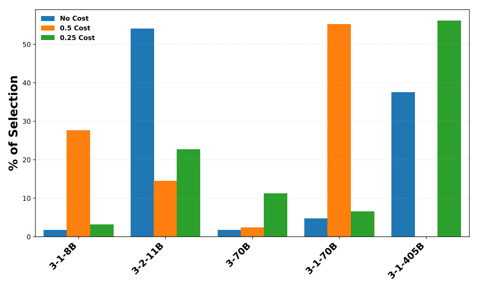
In this research question, we explore whether a model router can diversify model selections instead of consistently choosing a single model. While evaluating effectiveness and cost provides insights into a router model, it does not fully capture the router model's capabilities. For instance, a router might consistently select a model with lower cost than the highest-cost model, which is also effective across various tasks. This approach can lead to impressive effectiveness scores and cost values. However, the router may not excel in making optimal model selections.

For this research question, we primarily compare two techniques: the LLM Classifier and the LLM AR Model with GRPO. As we have observed that the LLM AR Model trained with SFT struggles to complete responses accurately, we have excluded these models from this evaluation. Figure 4 illustrates the percentage of calls made to different models using both routing techniques under various settings.

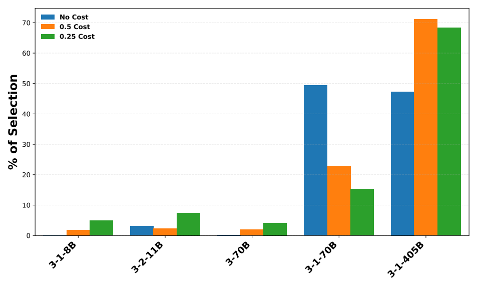
The figure reveals that LLM Classifiers fine-tuned with both Qwen-2.5-0.5B and Qwen-2.5-1.5B architectures do not consistently select specific models for all prompts. However, the diversity of the model fine-tuned on Qwen-2.5-0.5B is superior. For both models under Setting 1 (with zero cost), the majority (37-38%) of prompts predict the 3-1-405B model. For Qwen-2.5-0.5B, the lowest percentage of selections is for 3-1-8B (9.8%), confirming that the prediction is unbiased. Conversely, for Qwen-2.5-1.5B, the 3-1-8B model is predicted for only 1.7% of the total prompts.



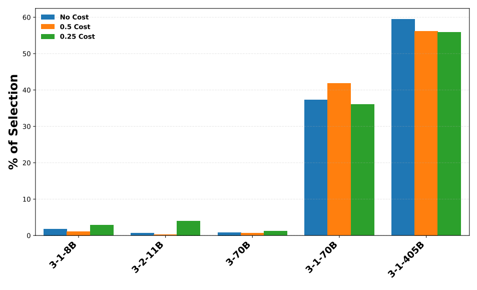
(a) LLM Classifier Qwen-2.5-0.5B



(b) LLM Classifier with Qwen-2.5-1.5B



(c) LLM-GRPO with Qwen-2.5-0.5B



(d) LLM-GRPO with Qwen-2.5-1.5B

Figure 4: Distributions of the model selections on LLM-GRPO and LLM classifier routers. The Blue, Green, and Orange bars represent setting 1, setting 2 and setting 3 respectively.

The LLM Classifier with Qwen-2.5-1.5B demonstrates improved diversity in Settings 2 and 3. In these settings, the classifier selects three models with more than 10% of the selection percentage. For the LLM Classifier with Qwen-2.5-0.5B, all four models other than 3-1-405B achieve higher than 10% of selections in these settings.

In Settings 2 and 3, we anticipate the router to predict models with lower costs. Therefore, we incorporated the cost component (0.25 and 0.5, respectively) into the training. In these settings, the LLM Classifier with Qwen-2.5-0.5B significantly reduces the selection of 3-1-405B models (8.8% and 0.7%, respectively, for Settings 2 and 3) while maintaining diversity. For the LLM Classifier with Qwen-2.5-1.5B, there is a notable reduction in predictions of the 3-1-405B model for Setting 3. However, for Setting 3, this pattern is not consistently observed.

For the LLM AR Model fine-tuned with GRPO, there is a discernible bias against certain models. The results indicate that two models, 3-1-405B and 3-1-70B, are predominantly selected. Additionally, we observed that in cost-sensitive settings, the selection of the 3-1-405B model slightly decreases for the Qwen-2.5-1.5B architecture. Conversely, for the Qwen-2.5-0.5B architecture, the selection of the 3-1-405B model increases instead of decreasing in cost-sensitive settings. This suggests that altering cost settings may not be effective for the LLM AR Model fine-tuned with GRPO.

A.4 Generalization to Unseen Tasks

To test the generalizability of our trained router to novel tasks, we use the BigCodeBench dataset [Zhuo et al., 2025], consisting of 1k synthetic code generation tasks reliant on common Python libraries. To avoid re-running base Llama models, we leverage pre-computed model predictions bundled with the dataset. Llama 3-70B as well as Llama 3.1 8B and 70B are available. In the best effort to match the missing Llama 3.2 11B and Llama 3.1 405B, we use CodeLlama 13B [Rozière et al., 2024] and Claude Sonnet 3.5, as they represent similar performance tiers to the originally used models. Router model trained on our curated datasets is applied to BigCodeBench as-is, without further finetuning. We note, a potential mismatch with the task in the domains of this datasets and the ones used for training (e.g., the library use aspect). The base model pass@1 scores on this dataset range from 0.27 (CodeLlama-13B) to 0.51 (Claude Sonnet 3.5). The overall success rate for the reference solutions was at 0.84 due to some dependency-related errors. We used the same cost multipliers as with the original models. Results are shown on Figure 5. As before, the Oracle routing sets the upper bound of effectiveness at 0.63 and the lower bound of cost at 0.078. In the cost-free setting, the CLS router achieves within 1% absolute of the strongest model, however the cost is nearly as high, at 91%. With the cost weight of 0.25, CLS achieves a more favorable trade-off at 0.4877 effectiveness and 0.7 cost. It should be noted, however the next best performing base model Llama 3.1 70B is only less than 1% behind at one third of the cost. Finally, with the cost weight of 0.5, CLS outperforms Llama 3.1 8B by 1.3% while costing 7.8% vs 1.3%.

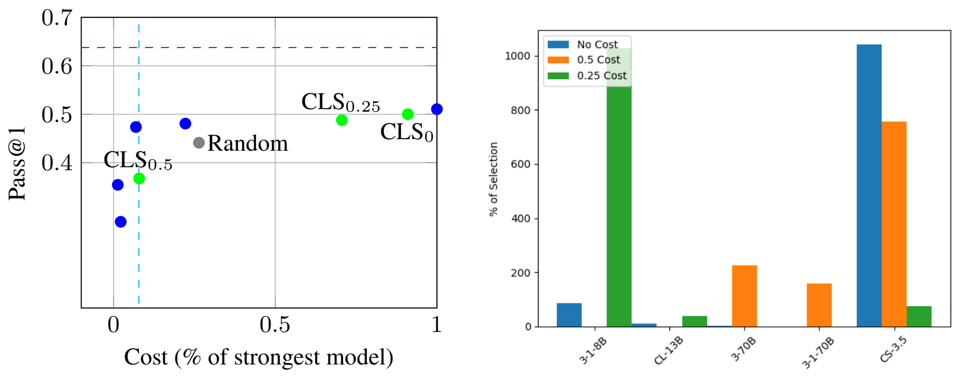


Figure 5: Left: Cost-performance trade-off of different of the CLS-based routing on BigCodeBench. Unlabeled blue circles represent the 5 Llama3 LLMs of different sizes. Subscript under CLS indicates cost weight used. Vertical and horizontal dashed lines represents oracle's cost and performance, respectively. Right: Distribution of model routing decisions across the 3 cost settings.

A.5 Overview of Finetuning Techniques Used

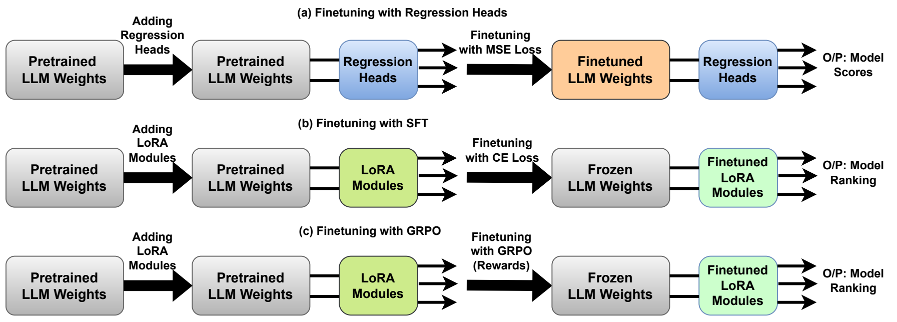


Figure 6: Different Finetuning Techniques Used

A.6 GRPO Reward Formulation

Algorithm 1 GRPO Reward Function

```
Require: LLM outputs Outputs, ground-truth answers Answers
Ensure: Rewards list
1: Define regex pattern for comma-separated model names
2: for each (response, target) in Outputs, Answers do
3:     if response matches pattern then
4:         pred = first token in response split by comma
5:         target_list = labels from target split by comma
6:         Assign reward $r$ based on pred position in target_list: $1.0, 0.5, 0.25$, or $0.0$
7:     else
8:         Assign reward $r = -5.0$
9:     end if
10:    Append $r$ to rewards
11: end for
12: return rewards
```

A.7 Additional Details on Dataset and Normalization

As discussed in Section 4, we selected five datasets for fine-tuning and evaluating different techniques: SVEN, APPS, RunBugRun, Code Refinement, and Code Translation. Although we initially started with a higher number of samples for each dataset, the collected data points have been reduced due to failed executions. Table 3 presents the breakdown of training and test data sizes for each dataset. Although we initially started with a higher number of samples for each dataset, the collected data points have been reduced due to failed executions.

Given the significant differences in datasets in terms of value ranges and sample numbers, we apply the following transformations before training. We use standard scaling by subtracting the mean and dividing by the standard deviation to center samples around the mean and normalize their range to (-1.0, 1.0).

For binary tasks like SVEN, we add an additional penalty score for fine-tuning. While generating binary labels through LLMs can be straightforward, the biased behavior of an LLM can make effectiveness calculation error-prone. For instance, in a dataset with 50% positive and 50% negative true values, if an LLM classifies every example as positive, it would be correct 50% of the time despite being incapable. To address this, we use a penalty term for calculating effectiveness. The penalty P can be calculated as $P = \frac{|N_{tp} - N_{pp}|}{N_t}$ where, N_{tp} is the number of true positive examples, N_{pp} is the number of predicted postive examples and N_t is the total number of examples. Usage of absolute values makes sure that if the model is biased towards the negative examples, the penalty would still be high. This penalty term is added on each effectiveness score for this task where the penalty value would be different for different models.

Table 3: Number of training and test data from each dataset

# Samples	RBUGR	APPS	REFI	TRANS	SVEN
Train	1145	2824	3750	1125	483
Test	381	941	1250	375	161

A.8 Prompt Types and Structures

A.8.1 SVEN

```
Tell me if the following python code has vulnerability or not.
    Use only Yes or No.

# Code
```

A.8.2 RunBugRun

```
Provide a fix for the buggy function.

Buggy Function:

# Code

Assertion Error:

# Assertion Error
```

A.8.3 APPS

```
Given the below programming question and example inputs/outputs
    , write Python code that solves it. Your program should be
    runnable inline and accept arguments via standard input and
     print back the result to standard output.

Question:

# Question
```

A.8.4 Code Translation

```
Task: Translate the following code from Java to C#.

# Code

Translate this code to C#, maintaining the same functionality
    and following C#'s best practices.
```

Code Refinement

```
Task: Remove the bugs in the following code.

Original Code (All the function and variable names are
    normalized):

# Code

Provide a refined bug-free version.
```

A.8.5 System Prompt for Routing (w Autoregressive Models)

```
Given a task and code snippet, return the model ranking based
    on the expected effectiveness to execute the prompt.

There are five models and each model represented by a number
    from 0 to 4. The numbers must be distinct and sorted in any
     order. Example valid outputs include: "0,1,2,3,4",
    "4,0,1,3,2", or "2,3,1,0,4". Do not include any extra text,
     explanation, or punctuation beyond the numbers and commas.
     The output must always contain exactly five numbers.
```