

STARouter: Internal State based LLM Router for Software Testing Tasks

Hyunjoon Cho
School of Computing
KAIST
Daejeon, Republic of Korea
hyunjoon.cho@kaist.ac.kr

Shin Yoo
School of Computing
KAIST
Daejeon, Republic of Korea
shin.yoo@kaist.ac.kr

Abstract—As Large Language Models (LLMs) are rapidly adopted for software engineering tasks, practitioners face growing needs to choose a model that fits their tasks. Such a choice may not be easy, as models provide conflicting benefits. Small Language Models (SLMs) may be cheaper, more secure, and have lower latency if they are run locally, but they may also be limited in their capabilities. Closed source LLMs that are served remotely are much stronger, but they incur monetary cost, have high latency, and raise privacy or security concerns. We introduce STARouter, an LLM router for software engineering tasks that can automatically choose the appropriate model between local and remote LLMs. STARouter can be trained using inputs for a specific task, each labelled with whether two models succeed or fail, respectively. Using such a dataset, STARouter learns to recommend the model that would successfully achieve the given task, with a preference for the local model, based on the internal state of the model after processing the given input. We empirically evaluate STARouter against five testing tasks and two code generation tasks, using five open source SLMs and two closed source LLMs. Results shows that STARouter can effectively route inputs for LLMs, recovering 80% of the performance gap between local and remote models while routing as small as 55% of inputs to the remote models.

Index Terms—Large Language Model, Software Testing, LLM Routing

I. INTRODUCTION

Large Language Models (LLMs) are being rapidly adopted by software engineers as a part of software development process, such as code and test case generation [1], thanks to the advances in emergent capabilities shown by the state-of-the-art models. Since such advanced LLMs are usually served online over network in commercial settings, such integration of LLMs into development process typically faces two challenges. First, querying state-of-the-art LLMs may entail security risks by transmitting internal assets (such as existing code) to the model provider. Second, use of commercial LLMs incur financial cost that is linear to the number of tokens used by the queries. To overcome these challenges, Small Language Models (SLMs) with open model weights are gaining traction [2]: they are small enough to be served locally, and usually significantly cheaper to use. However, there also exists performance gap between state-of-the-art LLMs and SLMs, especially depending on tasks and prompts [3].

Language model call optimization encompasses techniques to address this emerging issue, most notably by choosing

the right model considering both the cost and the expected performance of each model. Depending on the point of model selection, i.e., before or after output generation, call optimization techniques are categorized into routing and cascading [4]. Routing aims to choose the right model first and starts the generation of output, whereas cascading starts generating the output with the default model and defers the task to another model only when the generated output is deemed unacceptable.

While cascading tends to outperform routing in general Q&A [4], [5], its application to software engineering is hampered by the complexity of quality estimation and the costs of initial generation. This motivates our focus on routing. Existing routers typically rely on off-the-shelf embeddings, which often lack domain-specific nuance [6], or fine-tuned SLMs, which require expensive training and high-quality data [7]. To address these limitations, we introduce internal STATE based Routing (STARouter), which utilizes hidden states from the default SLM as the routing signal. By capturing the model's own semantic representation of a prompt, STARouter enables efficient routing via lightweight classifiers (e.g., Logistic Regression or Multi-Layer Perceptron) without the overhead of output generation or specialized embedding models.

We empirically evaluate STARouter using software testing and code generation benchmarks, investigating whether this state-based routing can choose the optimal model for each task. Since there is no existing router model specifically designed for software engineering tasks, we use RouteLLM [6], a routing technique designed to optimize LLM calls for human preferences, as a baseline, and measure how optimal the routing results are. The results show that it is possible to recover 80% of the performance of the strong, remote model, while routing as small as 55% of calls to the remote model and processing the remaining calls with the weaker local SLMs.

Our key technical contributions are as follows:

- We propose STARouter, an internal-state based LLM routing technique that achieves greater routing performance than embedding based approach.
- We introduce a new evaluation metric, Router Optimality (RO), which accurately measures routing performance without being confounded by model performance.
- We empirically evaluate STARouter using widely-used software engineering LLM benchmarks. STARouter

continuously outperforms the prompt embedding-based router on both testing and code generation tasks. We also evaluate the generalization of STARouter under model and benchmark shifts.

The rest of the paper is organized as follows. Section II presents background information about language model call optimization. Section III describes how STARouter performs routing based on internal states taken from SLMs, while Section IV explains how we analyze the optimality of routing. Section V describes the settings of the empirical evaluation, the results of which is presented in Section VI. Section VII lists threats to validity. Finally, Section VIII concludes.

II. BACKGROUND

As a set of diverse language models, each with different size and capability, becomes available, it is increasingly important to select the appropriate model for the given task. In particular, the selection needs to be dynamic, so that we rely on larger and more expensive models only when they are strictly required.

Ramirez et al. outline two major approaches for this problem: routing, in which we make decisions *before* generating any tokens, vs. cascading, in which decisions are made *after generation* [4]. In cascading, an SLM generates a response first, based on which we decide whether to query a more capable model. Cascading approaches often yield a superior cost-performance trade-off [4], [5], as the initial generation results provide extra information for better accuracy.

In routing, the decision of which model to call is made before any output generation, based solely on the task input. Either the embedding vectors of the task input itself, or some semantic representation of it, can be used as the input to the decision of routing. HybridLLM [7] adopts BERT-based encoders and fine-tunes them to estimate the performance gap between LLMs and SLMs, based on which the routing decision is made. More recently, RouteLLM [6] reformulates the routing problem to predict a win probability instead of the performance disparity. Building on human preference labels from the Chatbot Arena dataset [8], they propose various forms of routers – from a prompt embedding-based statistical model to a fully fine-tuned small language model – to model the preference and to route.

The focus of cascading is to efficiently assess the quality of the output generated by the first model. FrugalGPT [9] proposes to train a separate scoring function to choose when to stop relaying the query to the next candidate model. While this approach can achieve acceptable cost-performance trade-off, the scoring function is highly dependent on the specific task. Further, cascading incurs extra generation cost proportionate to the computational complexity of the models involved.

Uncertainty-based cascading tackles this initial cost and the task-specificity by only generating the very first token, then capturing the margin between the two most probable tokens as the uncertainty [4]. While this was shown to be effective against relatively short outputs, we suspect that the uncertainty captured from the first token may be too weak a signal for software engineering tasks, in which the correctness of the

final output is a global, context-sensitive property. As the generated code must adhere to grammatical constraints and identifier consistency, token-level uncertainty is an insufficient measure to reflect this global structure.

III. STAROUTER: INTERNAL STATE BASED ROUTING

This section formulates the LLM routing problem, then details the design of STARouter, beginning with its input representation based on LLM internal states. Subsequently, we discuss labeling setup for the strong model, M_{strong} , vs. the weak model, M_{weak} , as well as the choice of router model.

A. Problem Formulation

We adopt the formulation of RouteLLM [6], and train a router to predict the *win* probability of a model given two language models with performance disparity. We specifically assume that the open-source SLM is deployed locally while the proprietary LLM is accessed via API calls. We denote each model as the *Weak* model M_{weak} and the *Strong* model M_{strong} . We define a *win* for M_{weak} as cases where the *Weak* model achieves equal or better performance on the given query compared to the *Strong* model. For a binary task – such as generating a function that passes provided testcases, if both models fail to pass the testcases, we regard it as a win for the *Weak* model. This comparative definition clearly distinguishes routing from absolute difficulty modeling, as we prioritize cost-efficiency when both models fail.

Given these two models, the router, denoted as a function R , takes a task query q and a win probability threshold τ . A binary classifier predicts the probability that the *Weak* model will win the *Strong* model on the given task. Let $f(q) \in [0, 1]$ denote this probability. Then, we compare this value to the threshold to make a routing decision: if the win probability is greater than or equal to the threshold, we route this query to the *Weak* model. Otherwise, we generate a solution using the *Strong* model. This function can be described as follows:

$$R(q, \tau) = \begin{cases} 1 \text{ (route to } M_{\text{weak}}), & \text{if } f(q) \geq \tau \\ 0 \text{ (route to } M_{\text{strong}}), & \text{otherwise} \end{cases}$$

Our goal is to accurately calibrate win probabilities so that we can effectively identify samples that are unsolved by the *Weak* model, while avoiding unnecessary calls to M_{strong} for queries that are already handled by M_{weak} or are unsolvable for both models.

B. Internal State

To route an input, we extract the internal state of the weak model, M_{weak} , when processing the last token of the input query. Considering the masked self-attention mechanism adopted by modern decoder-only language models, the representation of the last token attends to all previous tokens, and thus reflects the information from the entire prompt.

We argue that extracting the internal state of the last token from M_{weak} , instead of embedding the entire input prompt, is a practical design choice for a routing technique. Compared to embedding models, we expect the internal state to contain

richer information. The cost of extraction may be higher than that of embedding the input prompt, but remains significantly lower than actually generating the output for cascading so the trade-off may still be attractive. Further, if a query is routed to the SLM during deployment, generation can commence without redundant prefilling, making the practical overhead smaller than that of querying an external embedding model.

Among multiple layers of the MLP block in LLM decoders, we specifically focus on middle to late layers: for example, between the 16th and the 24th layers out of 32 total layers of Llama3 [10]. This is based on findings that middle to late layers focus on abstract information, whereas the final layers focus on aggregating such information to predict the probability distribution of the actual token [11]. A recent work shows that middle to late layers contain signals of task success [12], making them ideal inputs of STARouter.

C. Labeling

To train the router model, we label each sample for every model pair based on the performance of M_{weak} and M_{strong} . Since all our target tasks provide automated evaluation scheme for the generation result, we directly compare the outcomes of the two models on each problem.

We repeat each input query 10 times before labeling the input, catering for stochastic nature of LLM inferences. All studied benchmarks provide automated oracles that check the correctness of the generated output. This, in turn, allows us to compute pass rate out of 10 queries for the given task. After comparing the pass rates from M_{weak} and M_{strong} , if one model achieves higher pass rate, then we label the input with the corresponding model. If both models produce the same result, we label the input with M_{weak} because it would be more cost efficient to choose M_{weak} for the same output.

D. Routing Model

Considering that labeling training data for a routing technique involves making queries to both M_{weak} and M_{strong} , it is important that the routing model can be robustly trained with smaller training datasets. Even for evaluation, the largest benchmark we study contains fewer than 1.5K inputs, reflecting the cost of collecting and processing such data (in comparison, RouteLLM trains its router using 80K input corpus [6]). Since internal states extracted from SLMs have dimensions greater than 3,000, training on limited data is susceptible to overfitting.

For this reason, we employ lightweight machine learning algorithms as our routers: Logistic Regression (LR), Support Vector Machine (SVM), Multi-Layer Perceptron (MLP), and an ensemble of these three models. For LR and SVM, we apply Principal Component Analysis (PCA) to the input to reduce its dimension so that the input dimension remains lower than the number of training samples. For MLP, we set the hidden layer size low to prevent overfitting.

IV. ROUTER OPTIMALITY METRIC

As routers tackle the cost-performance trade-off under the scenario of using two or more language models, its perfor-

mance is often visualized as a curve whose x -axis is the cost of LLM calls and y -axis is the target task performance. Ramirez et al. [4] use Area Under the Curve (AUC) computed from this curve to quantify the performance of a call optimizer. While AUC is effective for general classification, it is highly affected by the task performance of the candidate models, making it difficult to assess the effectiveness of router itself. Here, we present existing evaluation metrics for LLM routing as well as their limitations, and present Router Optimality (RO), our own quantitative performance metric for LLM routing. Since we assume that the weaker model, M_{weak} , is deployed locally, whereas the stronger model, M_{strong} , is remote and incurs API call costs, we simplify the cost as the proportion of calls to M_{strong} , focusing on the cost of making calls to M_{strong} .

A. Existing Metrics

1) *Average Performance Gap Recovered*: RouteLLM [6] proposes *Average Performance Gap Recovered* (APGR) to assess routing performance more accurately. Suppose that we are working on a benchmark B of size N , and also that each query is denoted as q_i . Let $p_{s,i}$ and $p_{w,i}$ be the performance (e.g., 1 if solved 0 otherwise) of *Strong* and *Weak* models on i -th input of the benchmark. The aggregate performance of each model on the benchmark would be $P_s = \frac{1}{N} \sum_{i=1}^N p_{s,i}$ and $P_w = \frac{1}{N} \sum_{i=1}^N p_{w,i}$, respectively.

Now, we define the *Performance Gap* as the difference between these aggregate performances, $P_s - P_w$. Now, let $P_{R,\tau}$ be the aggregated performance for a router R at a specific threshold τ . For this mixed query setup, we add up performances of each model based on the routing decisions. This can be expressed as:

$$P_{R,\tau} = \frac{1}{N} \sum_{i=1}^N [(1 - R(q_i, \tau))p_{s,i} + R(q_i, \tau)p_{w,i}]$$

We then compute *Performance Gap Recovered* (PGR), to see how much of the performance gap is restored by routing:

$$PGR = \frac{P_{R,\tau} - P_w}{P_s - P_w}$$

For $\tau \in [0, 1]$, we compute PGR and proportion of calls routed to M_{strong} . We then define APGR as the area under the curve of PGR plotted against this call proportion, capturing the routing performance across the full spectrum of costs.

While APGR focuses more on the routing performance, it is not without shortcomings. Critically, APGR implicitly assumes that P_s is the theoretical upper bound. In practice, an *Oracle* router – one that perfectly selects the best model for every query – would outperform the sole use of M_{strong} as long as M_{weak} can complement M_{strong} , i.e., if it can solve problems that M_{strong} cannot.

2) *Call Performance Threshold*: *Call Performance Threshold* (CPT) is another metric suggested in RouteLLM [6] to directly measure the cost savings from the router. Given a target PGR $x\%$, we define $\text{CPT}(x\%)$ as the minimum proportion of calls routed to M_{strong} . Due to the discrete nature

of finite evaluation sets, this metric can exhibit overestimation in cases where the predicted probabilities are poorly separated or clustered. Nevertheless, we adopt CPT as a complementary metric as it effectively quantifies the cost efficiency.

Figure 1a visualizes PGR and CPT within a synthetic router cost-performance curve. As PGR at each data point on the Router curve corresponds to the vertical distance from the Weak Only line normalized by the *Performance Gap*, APGR becomes the ratio of the green area B to the blue area A.

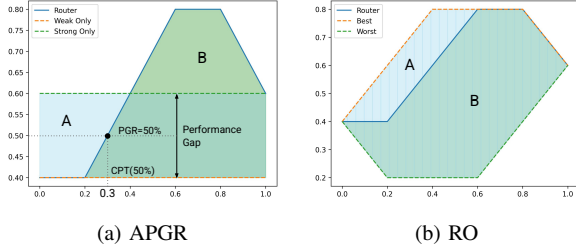


Fig. 1: Visualization of Average Performance Gap Recovered, Call Performance Threshold, and Router Optimality.

B. Definition of Router Optimality

We introduce a new metric that measures LLM routing performance, called *Router Optimality*, which is a normalized measure bounded strictly between the worst and best possible routing strategies.

We start by defining the gain from routing a query from M_{weak} to M_{strong} . If we choose to query M_{strong} for the i -th input, the aggregate performance gain would be $\delta_i = \frac{1}{N}(p_{s,i} - p_{w,i})$. Depending on the task and the language models, δ_i could be positive, negative, or zero. Since we want to maximize these gains while keeping the overall cost minimal, an optimal router would first route all inputs with positive gain, zero gain samples next, and finally route negative gain samples to M_{strong} . For evaluation purpose, assuming that we know the outcomes of all inputs when sent to both M_{weak} and M_{strong} , we can simulate the optimal routing by sorting inputs in the descending order of their performance gain. The worst case scenario is the exact opposite, where we sort samples in ascending order of their gains.

We can transform these best and worst orderings into a cost-performance curve by computing aggregate performance as we increase the number of queries routed to M_{strong} from zero to N . Let these curves for the best and worst scenarios be C_b and C_w . And, let the curve drawn from a router be C_r . Then, we define a function AUC that takes a curve C as input and computes the area under C . Then, $\text{AUC}(C_w)$ and $\text{AUC}(C_b)$ are the areas corresponding to the theoretical worst and best routing strategies, respectively. We define RO as the ratio between area A and B in Figure 1b:

$$\text{RO} = \frac{\text{AUC}(C_r) - \text{AUC}(C_w)}{\text{AUC}(C_b) - \text{AUC}(C_w)}$$

Router Optimality can be interpreted as a normalized position of the actual router's curve between these best and worst envelopes. Unlike APGR, which normalizes against a static performance gap, RO accounts for the dynamic range of possible routing outcomes, rewarding the router for correctly prioritizing high-gain samples.

V. EXPERIMENTAL SETUP

A. Research Questions

Our empirical evaluation is designed to answer the following research questions:

RQ1. Effectiveness: To what extent does our router approximate the optimal routing scenario? We quantify the router's utility in software engineering domains, while evaluating the hypothesis that the internal states provide more discriminative features for routing compared to the off-the-shelf embeddings.

- **RQ1-1.** Do internal states extracted from small language models serve as better routing inputs than embeddings from off-the-shelf models?
- **RQ1-2.** What is the trade-off between cost savings and performance retention when querying M_{strong} models?

RQ2. Generalizability: Does our approach generalize across different contexts? We investigate the cross-model and cross-domain transferability of the routing logic. As the internal states are extracted from a specific SLM, we characterize the degree of coupling between the router and its source SLM by comparing its degree of generalization to that of prompt embeddings.

- **RQ2-1.** Does a router trained on one pair of *Strong* and *Weak* models transfer effectively to other model pairs?
- **RQ2-2.** Does a router trained on one benchmark maintain performance on unseen benchmarks?

B. Datasets

Table I summarizes the benchmarks used to evaluate the router. For testing, we employ TestEval [13], a test generation benchmark for LLMs, and LIBRO [14], a bug reproduction benchmark based on Defects4J [15].

TestEval consists of four subtasks: (1) total, which involves multi-turn test generation to maximize overall coverage, and targeted generation tasks where the model must cover a specific (2) line, (3) branch, and (4) path. Targeted path constraints describe the execution flow explicitly (e.g., 'Line 26: (j in range(n))' -> 'Line 27: NOT (p[j]=='*')' -> 'Line 31: (isMatch(i, j))' -> ... -> 'Line 25: (i in range(m))'). LIBRO [14] aims to generate test cases that reproduce the natural language bug report contained in Defects4J [15]. We set temperature 0.0 for TestEval tasks, while setting it 0.7 for the case of LIBRO on Defects4J, each following its original evaluation setup.

We include two widely used code generation benchmarks to understand the relative difficulty of routing test generation tasks: APPS [16] and HumanEval [17]. APPS is sourced

TABLE I: Overview of Benchmarks

Category	Benchmark	Task Description	Metric	Samples
Testing	LIBRO(Defects4J)	Generate bug reproducing test given an issue related to the Java bug.	<i>reproduced@10</i>	744
	TestEval(total)	Generate multiple unit tests given a target Python function to maximize overall coverage. (N=10)	<i>line-cov@1</i>	210
	TestEval(line)	Generate a unit test targeting a specific line in the function.	<i>line-covered</i>	1340
	TestEval(branch)	Generate a unit test targeting a specific branch condition.	<i>branch-covered</i>	983
	TestEval(path)	Generate a unit test targeting a specific execution path .	<i>path-covered</i>	854
Code Generation	APPS	Generate Python code given a natural language description of the problems sourced from coding challenges.	<i>pass@10</i>	300
	HumanEval	Generate Python code from function signatures and docstrings.	<i>pass@10</i>	164

from multiple coding challenge websites, with three different difficulty levels - introductory, interview, and competition, containing 10,000 problems within it. We sample 300 problems, 100 from each level, from the test split. HumanEval is an early code generation benchmark which queries a language model to complete a total of 164 python functions, given signatures and docstrings. Evaluation for APPS and HumanEval uses a temperature of 0.0.

For all benchmarks, we use the original task evaluation metrics. For TestEval(total), we iteratively generate 10 test cases with each query and use the average coverage achieved by each single test case, *line-cov@1*, as the performance measure [13]. For line, branch, and path objectives of TestEval, each query generates a single test case, for which we use the binary metric of 1 (if target is covered) or 0 (not covered). Similarly, HumanEval, APPS, and LIBRO use *pass* or *reproduced@10*.

C. Language Models

We evaluate two proprietary models (M_{strong}), gpt-4o [18] and gemini-2.5-flash-lite [19], and five open-source models (M_{weak}), Llama3 and 3.1 8B [10], Mistral-NeMo 12B [20], Phi-4 14B [21], and Qwen2.5-Coder 7B [22]. We use transformers [23] library to serve as well as to extract internal state from SLMs. Based on preliminary results showing indistinguishable router performance across middle-to-late layers, we report all results using hidden states from the 3/4-th layer.

D. Embedding Models

We embed prompts using a proprietary embedding model, `text-embedding-3-large` from OpenAI. Through preliminary experiments, we find that low capacity embedding models often degrade router performance; we thus focus on this high-capacity model to ensure robust input representation.

E. Routing Models

We use Logistic Regression (LR), Support Vector Machine (SVM), and Multi-Layer Perceptron (MLP) from scikit-learn (v1.5.1) [24]. For the LR and SVM routers, we apply PCA to reduce the input to 24 dimensions. The MLP router consists of a single hidden layer, whose dimension is 32.

We adopt the Matrix Factorization (MF) architecture as our baseline from RouteLLM [6], as it is reported to outperform alternative designs for general QA dataset. We re-implement this model using PyTorch [25] framework, tuning

its hyperparameters to suit our dataset scale. While RouteLLM is originally designed to train a single global router across all language model pairs, we train a separate instance of the baseline for every model pair on each benchmark to ensure a fair comparison.

Given the limited dataset size compared to prior routing work, we adopt five-fold cross-validation rather than a static train/test split. To prevent routers from acquiring project or problem-specific patterns, we stratify the folds such that tasks derived from common sources – such as the same source project in Defects4J or the same parent problem in TestEval – are contained within a single fold.

F. Environment & Replication

We conduct all experiments on a Docker container of `nvidia/cuda:11.3.1-runtime-ubuntu20.04` which is deployed on a Linux server with 252 GB RAM and 40 Intel Xeon Silver 2.40GHz CPUs. For the serving of small language models, we use two NVIDIA GeForce RTX 3090 GPUs. The replication package for the empirical evaluation is available from <https://github.com/coinse/starouter>.

VI. RESULTS

A. RQ1. Effectiveness of the Router

Table II presents the Router Optimality (RO) statistics across router architectures and benchmarks. We observe two distinct trends regarding input representation and task type.

First, internal state-based routing consistently outperforms prompt embedding. A one-sided Wilcoxon signed-rank test confirms that performance gaps are consistently statistically significant ($p < 0.05$) for LIBRO and APPS. Internal state-based routing tends to outperform embedding-based routing for TestEval and HumanEval, but the results are more mixed and gaps are less pronounced. Notably, for TestEval(total), embedding-based routing outperforms internal state-based routing. We suspect this may be related to the cumulative nature of the task: TestEval(total) requires multi-turn interaction ($N=10$) unlike others, extending its prompt by appending previously generated testcases. While internal state specifically focuses on the immediate next testcase generation, prompt embedding likely captures only the global semantics of the target task. We believe this stable representation enables it to outperform internal states as a router input on TestEval(total).

TABLE II: Comparison of Router Optimality using Internal State vs. Prompt Embedding. Values represent Mean (Standard Deviation). Statistical significance is assessed using the one-sided Wilcoxon signed-rank test, $\mu_{\text{Internal State}} > \mu_{\text{Prompt Embedding}}$.

Benchmark	Input	LR		SVM		MLP		Ensemble		RouteLLM	
		RO	p-val	RO	p-val	RO	p-val	RO	p-val	RO	p-val
LIBRO(D4J)	Internal State	0.592 (0.039)	< 0.001	0.536 (0.058)	< 0.001	0.597 (0.024)	< 0.001	0.588 (0.029)	< 0.001	0.572 (0.052)	0.024
	Embedding	0.509 (0.039)		0.484 (0.041)		0.514 (0.040)		0.509 (0.041)		0.537 (0.038)	
TestEval(total)	Internal State	0.518 (0.061)	0.754	0.468 (0.055)	0.688	0.523 (0.045)	0.812	0.513 (0.066)	0.784	0.528 (0.061)	0.246
	Embedding	0.529 (0.077)		0.470 (0.050)		0.539 (0.052)		0.528 (0.068)		0.508 (0.039)	
TestEval(line)	Internal State	0.548 (0.043)	0.348	0.533 (0.033)	0.688	0.536 (0.048)	0.577	0.545 (0.050)	0.784	0.566 (0.046)	0.042
	Embedding	0.545 (0.036)		0.541 (0.037)		0.541 (0.043)		0.549 (0.039)		0.532 (0.034)	
TestEval(branch)	Internal State	0.560 (0.036)	0.002	0.540 (0.031)	0.014	0.544 (0.037)	0.246	0.564 (0.041)	0.007	0.556 (0.044)	0.019
	Embedding	0.530 (0.032)		0.507 (0.037)		0.536 (0.031)		0.537 (0.034)		0.533 (0.036)	
TestEval(path)	Internal State	0.496 (0.025)	0.246	0.497 (0.026)	0.652	0.511 (0.018)	0.042	0.506 (0.025)	0.065	0.501 (0.031)	0.065
	Embedding	0.486 (0.044)		0.493 (0.033)		0.484 (0.034)		0.480 (0.040)		0.487 (0.025)	
APPS	Internal State	0.782 (0.080)	< 0.001	0.772 (0.072)	< 0.001	0.717 (0.076)	0.002	0.784 (0.080)	< 0.001	0.721 (0.086)	< 0.001
	Embedding	0.690 (0.071)		0.677 (0.082)		0.660 (0.066)		0.696 (0.079)		0.666 (0.068)	
HumanEval	Internal State	0.565 (0.074)	0.003	0.542 (0.114)	0.539	0.588 (0.057)	0.577	0.583 (0.072)	0.246	0.587 (0.061)	0.423
	Embedding	0.526 (0.065)		0.541 (0.068)		0.587 (0.061)		0.566 (0.075)		0.584 (0.041)	

Additionally, all routers show near-random performance (RO ≈ 0.5) on TestEval(path). We conjecture that the *unnatural* format of execution paths makes it difficult for both internal states and embeddings to effectively represent the task.

We note that the simplest architecture, Logistic Regression, often outperforms more complex counterparts when using internal states. This suggests that internal states may offer naturally well-separated features, allowing simple linear decision boundaries to succeed where computation-heavy models (like MLP or RouteLLM) yield diminishing returns given the current sample size. In comparison, SVM tends to underperform across benchmarks, suggesting that it does not generalize as well as LR beyond training data.

Second, Code Generation tasks are routed more effectively than Testing tasks, as can be seen from the fact that RO scores for code generation are notably higher. We interpret that this gap reflects the task nature: code generation is often descriptive and self-contained, whereas test generation requires a deeper semantic understanding of program behavior, making the routing decision more ambiguous.

Table III further validates these findings through the lens of cost-efficiency, measuring the Call-Performance Threshold (CPT) required to recover 50% or 80% of the performance gap. Lower CPT values indicate greater cost savings. Consistent with the RO results, internal state-based routers generally require fewer calls to M_{strong} to meet performance thresholds. The task dependency is amplified here: to recover 50% of the performance gap, testing benchmarks require routing approximately 45% of queries to M_{strong} , whereas code generation tasks require approximately only 30%.

Answer to RQ1: We empirically validate that internal state-based routers predominantly outperform prompt embedding-based baselines, demonstrating an RO improvement up to 9.5 percentage points on the APPS benchmark.

B. RQ2. Generalizability

Table IV summarizes router optimality when a router trained on one model pair is evaluated on another. We denote whether the strong/weak models are retained (=) or replaced ($\neq$).

TABLE III: Comparison of Call-Performance Threshold using Internal State vs. Prompt Embedding. Values represent Mean (Standard Deviation). The lower the better.

Architecture	Testing		Code Generation	
	Internal State	Embedding	Internal State	Embedding
LR	0.412 (0.102)	0.456 (0.095)	0.323 (0.185)	0.363 (0.167)
SVM	0.474 (0.109)	0.496 (0.102)	0.318 (0.186)	0.359 (0.170)
MLP	0.416 (0.080)	0.464 (0.094)	0.306 (0.149)	0.344 (0.137)
Ensemble	0.409 (0.089)	0.469 (0.104)	0.308 (0.164)	0.344 (0.153)
RouteLLM	0.420 (0.091)	0.473 (0.079)	0.306 (0.153)	0.336 (0.137)

(a) PGR: 50%

Architecture	Testing		Code Generation	
	Internal State	Embedding	Internal State	Embedding
LR	0.729 (0.133)	0.755 (0.102)	0.549 (0.266)	0.630 (0.245)
SVM	0.787 (0.099)	0.792 (0.095)	0.559 (0.276)	0.617 (0.248)
MLP	0.739 (0.098)	0.758 (0.090)	0.586 (0.261)	0.622 (0.199)
Ensemble	0.732 (0.092)	0.754 (0.087)	0.550 (0.265)	0.602 (0.230)
RouteLLM	0.726 (0.118)	0.758 (0.077)	0.572 (0.241)	0.618 (0.217)

(b) PGR: 80%

While altering the routed models leads to a degradation in router optimality, we observe a distinct asymmetry: replacing M_{strong} impacts performance more severely than replacing M_{weak} . We attribute this to the nature of capability overlaps. Weak models typically solve a subset of the tasks that strong models can solve; therefore, the decision boundary is largely defined by the unique, complementary capabilities of the strong model. Swapping the strong model shifts this boundary significantly, resulting larger performance drop.

Furthermore, comparing input representations reveals a trade-off between specificity and stability. As we move from left to right of the table, gradually replacing routed models, prompt embedding-based routers catch up with the performance of internal state-based routers. We conjecture that, as internal states are derived from SLMs, they perform well when the source of the internal states and the labels are matched. On the other hand, as prompt embeddings are free from the model choice, they tend to be stable across model changes.

TABLE IV: Cross model generalization. Values represent Mean (Standard Deviation) Router Optimality for each generalization scenario. (=) and ($\neq$) indicate whether the router’s training models match or differ from the routed models.

Architecture	Weak (=) Strong (=)		Weak ($\neq$) Strong (=)		Weak (=) Strong ($\neq$)		Weak ($\neq$) Strong ($\neq$)	
	Internal State	Embedding	Internal State	Embedding	Internal State	Embedding	Internal State	Embedding
LR	0.543 (0.053)	0.520 (0.051)	0.523 (0.051)	0.510 (0.041)	0.522 (0.046)	0.515 (0.049)	0.504 (0.038)	0.507 (0.037)
SVM	0.515 (0.050)	0.499 (0.045)	0.510 (0.042)	0.498 (0.038)	0.511 (0.045)	0.506 (0.038)	0.507 (0.036)	0.506 (0.036)
MLP	0.542 (0.046)	0.523 (0.045)	0.528 (0.051)	0.513 (0.037)	0.519 (0.042)	0.518 (0.044)	0.506 (0.035)	0.508 (0.038)
Ensemble	0.543 (0.053)	0.521 (0.050)	0.526 (0.050)	0.511 (0.041)	0.521 (0.051)	0.515 (0.050)	0.507 (0.039)	0.507 (0.040)
RouteLLM	0.545 (0.053)	0.520 (0.039)	0.527 (0.046)	0.512 (0.037)	0.521 (0.048)	0.514 (0.036)	0.506 (0.037)	0.507 (0.036)

(a) Testing

Architecture	Weak (=) Strong (=)		Weak ($\neq$) Strong (=)		Weak (=) Strong ($\neq$)		Weak ($\neq$) Strong ($\neq$)	
	Internal State	Embedding	Internal State	Embedding	Internal State	Embedding	Internal State	Embedding
LR	0.674 (0.134)	0.608 (0.107)	0.646 (0.155)	0.589 (0.127)	0.613 (0.090)	0.592 (0.095)	0.583 (0.105)	0.572 (0.108)
SVM	0.657 (0.150)	0.609 (0.101)	0.634 (0.151)	0.588 (0.119)	0.597 (0.120)	0.615 (0.089)	0.574 (0.116)	0.593 (0.110)
MLP	0.652 (0.093)	0.623 (0.073)	0.614 (0.105)	0.593 (0.088)	0.606 (0.081)	0.594 (0.063)	0.566 (0.090)	0.563 (0.077)
Ensemble	0.683 (0.127)	0.631 (0.100)	0.648 (0.149)	0.603 (0.120)	0.617 (0.090)	0.607 (0.082)	0.579 (0.105)	0.578 (0.101)
RouteLLM	0.654 (0.100)	0.625 (0.069)	0.620 (0.119)	0.596 (0.091)	0.598 (0.075)	0.597 (0.072)	0.564 (0.083)	0.567 (0.088)

(b) Code Generation

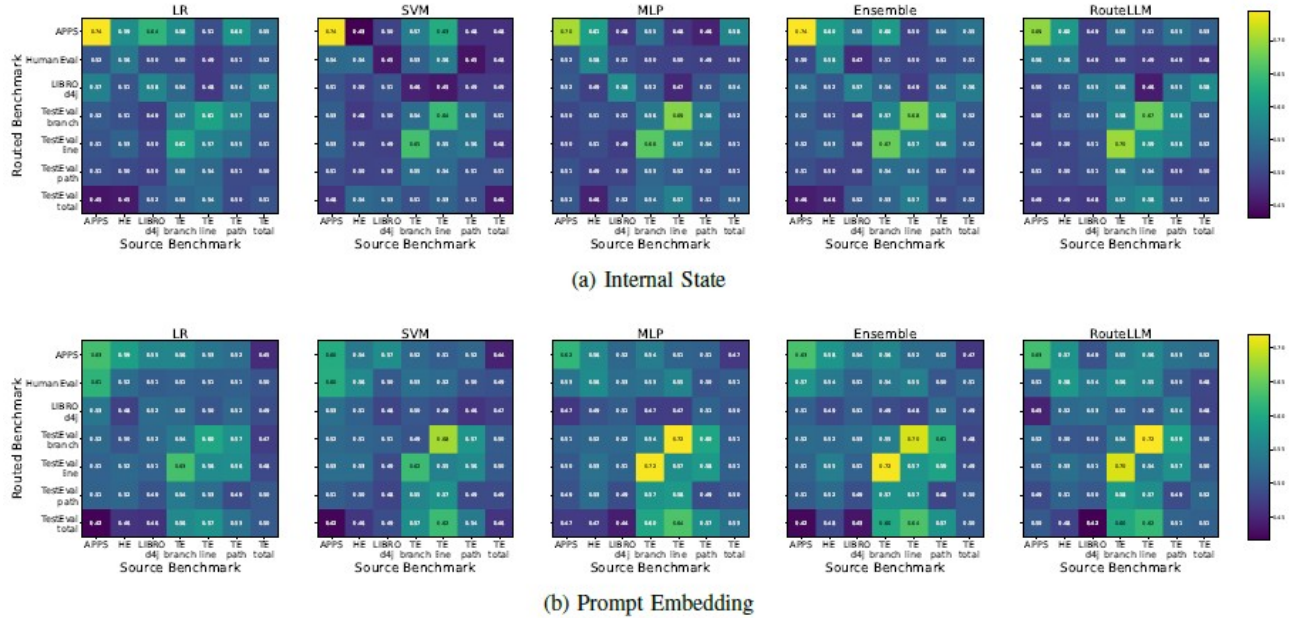


Fig. 2: Cross benchmark generalization. Each heatmap presents mean Router Optimality for the pair of benchmarks.

Figure 2 illustrates the RO results for the cross-benchmark generalization experiment. We fix the model tuple to focus on the cross-benchmark generalization trend. Unlike RQ1, we do not apply cross validation as we are training the router on one benchmark and testing on another. Diagonal elements represent the same benchmark performance from RQ1.

We observe high cross-compatibility between TestEval(line) and TestEval(branch). As these sub-benchmarks are derived from the same base problems, they exhibit high input similarity. Unlike the fold-based cross validation in RQ1 designed to prevent data leakage, this cross-benchmark setup naturally

leverages that similarity. We hypothesize this input consistency likely leads to label consistency, where problem difficulty remains stable across target coverage granularity.

The degree of generalization is also input-dependent. Heatmaps for prompt embeddings are broadly *lighter* - i.e., exhibit higher RO values than those for internal states. This supports the interpretation that embeddings are less bound to the specific internal patterns of a single SLM, though the high variance in internal state performance driven by APPS exaggerates this contrast. Additionally, routers that avoid PCA (MLP and RouteLLM) perform better, suggesting that PCA

may overfit to the principal components of the training distribution, hindering generalization to out-of-distribution inputs.

Answer to RQ2: Our results show that internal state-based routers generalize to OOD inputs, but exhibit greater variability than embedding-based approaches, potentially due to their coupling with the source SLM.

VII. THREATS TO VALIDITY

Threats to internal validity concerns any factor that may confound the relationship between independent and dependent variables. The benchmark based evaluation may be vulnerable to data leakage, but the purpose of model routing mitigates any threats from data leakage as STARouter works *between* different models. However, there exist threats to external validity that may limit the extent of generalization of our findings. We tried to mitigate threats to external validity by including multiple code related benchmarks, but further studies are needed to ensure that model routing can be done for a wider range of tasks. Threats to construct validity concerns whether we are measuring what we intend to measure. We adopt benchmark specific measures, as well as standardized prediction metrics, to minimize such threats.

VIII. CONCLUSION

We present STARouter, an LLM routing technique that uses the internal layer of the local, weak model to decide whether the given query should be sent to the remote, strong model. We also introduce a new evaluation metric for LLM routing that is normalized against varying performance of participating models. An empirical evaluation of STARouter with test case generation and code synthesis benchmarks shows that STARouter can achieve up to 78% of optimal routing using lightweight machine learning models as the router, and requires only 55% of inputs to be routed to the remote model in order to retain 80% of the performance gap between the weak local model and the strong remote model. Future work will investigate whether routing is possible for much more demanding workloads with longer interactions handled by LLM based agents, and also explore different input query representation that can provide even richer information.

ACKNOWLEDGMENT

This work has been supported by the National Research Foundation of Korea grant (RS-2026-25470171), the Engineering Research Center Program (RS-2021-NR060080), and the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant (RS-2022-II220995), all funded by the Korean Government (MSIT).

REFERENCES

- [1] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta *et al.*, "Large language models for software engineering: Survey and open problems," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering: Future of Software Engineering*, ser. ICSE-FoSE, May 2023, pp. 31–53.
- [2] F. Wang, Z. Zhang, X. Zhang, Z. Wu, T. Mo *et al.*, "A comprehensive survey of small language models in the era of large language models: Techniques, enhancements, applications, collaboration with llms, and trustworthiness," *ACM Transactions on Intelligent Systems and Technology*, vol. 16, no. 6, pp. 1–87, 2025.
- [3] N. Sinha, V. Jain, and A. Chadha, "Are small language models ready to compete with large language models for practical applications?" in *Proceedings of the 5th Workshop on Trustworthy NLP (TrustNLP 2025)*, 2025, pp. 365–398.
- [4] G. Ramirez, A. Birch, and I. Titov, "Optimising calls to large language models with uncertainty-based two-tier selection," *arXiv preprint arXiv:2405.02134*, 2024.
- [5] J. Dekoninck, M. Baader, and M. Vechev, "A unified approach to routing and cascading for llms," *arXiv preprint arXiv:2410.10347*, 2024.
- [6] I. Ong, A. Almahairi, V. Wu, W.-L. Chiang, T. Wu *et al.*, "Routellm: Learning to route llms with preference data," *arXiv preprint arXiv:2406.18665*, 2024.
- [7] D. Ding, A. Mallick, C. Wang, R. Sim, S. Mukherjee *et al.*, "Hybrid llm: Cost-efficient and quality-aware query routing," *arXiv preprint arXiv:2404.14618*, 2024.
- [8] W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li *et al.*, "Chatbot arena: An open platform for evaluating LLMs by human preference," *Proceedings of the 41st International Conference on Machine Learning*, vol. 235, pp. 8359–8388, Jul. 2024.
- [9] L. Chen, M. Zaharia, and J. Zou, "Frugalgpt: How to use large language models while reducing cost and improving performance," *arXiv preprint arXiv:2305.05176*, 2023.
- [10] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.
- [11] K. Meng, D. Bau, A. Andonian, and Y. Belinkov, "Locating and Editing Factual Associations in GPT," *Advances in Neural Information Processing Systems*, vol. 35, pp. 17 359–17 372, Dec. 2022.
- [12] J. Yoon, S. Kim, R. Feldt, and S. Yoo, "Clotho: Measuring task-specific pre-generation test adequacy for llm inputs," *arXiv preprint arXiv:2509.17314*, 2025.
- [13] W. Wang, C. Yang, Z. Wang, Y. Huang, Z. Chu *et al.*, "TestEval: Benchmarking Large Language Models for Test Case Generation," in *Findings of the Association for Computational Linguistics: NAACL 2025*, Apr. 2025, pp. 3547–3562.
- [14] S. Kang, J. Yoon, and S. Yoo, "Large Language Models are Few-Shot Testers: Exploring LLM-Based General Bug Reproduction," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23, Jul. 2023, pp. 2312–2323.
- [15] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, Jul. 2014, pp. 437–440.
- [16] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora *et al.*, "Measuring coding challenge competence with apps," *arXiv preprint arXiv:2105.09938*, 2021.
- [17] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.
- [18] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh *et al.*, "Gpt-4o system card," *arXiv preprint arXiv:2410.21276*, 2024.
- [19] G. Comanici, E. Bieber, M. Schaeckermann, I. Pasupat, N. Sachdeva *et al.*, "Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities," *arXiv preprint arXiv:2507.06261*, 2025.
- [20] Mistral AI team, "Mistral nemo," <https://mistral.ai/news/mistral-nemo>, 2024, accessed: 2026.
- [21] M. Abdin, J. Aneja, H. Behl, S. Bubeck, R. Eldan *et al.*, "Phi-4 technical report," *arXiv preprint arXiv:2412.08905*, 2024.
- [22] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu *et al.*, "Qwen2.5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.
- [23] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue *et al.*, "Transformers: State-of-the-Art Natural Language Processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Oct. 2020, pp. 38–45.
- [24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion *et al.*, "Scikit-learn: Machine learning in python," *The Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.
- [25] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury *et al.*, "PyTorch: An imperative style, high-performance deep learning library," *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, no. 721, pp. 8026–8037, Dec. 2019.