Vertical Routing: A Cost-Efficient Collaboration Routing Framework

Si Shen[1,2] and **Peijun Shen**[1] and **Danhao Zhu**[3,*]

[1]School of Computer Science and Engineering, Nanjing University of Science and Technology
[2]School of Economics and Management, Nanjing University of Science and Technology
[3]Jiangsu Police Institute
shensi@njust.edu.cn 124106022972@njust.edu.cn zhudanhao@jspi.cn
*Corresponding author

Abstract

Routing Large and Small Language Models (LLMs & SLMs) is commonly framed as query-level difficulty prediction, yet lightweight routers are often unreliable and stronger evaluators introduce non-trivial overhead. We propose VERTICAL ROUTING, a stage-level collaboration framework that avoids monolithic difficulty prediction by allocating the large model to *critical* subtasks and delegating the remaining generation to a small model. VERTICAL ROUTING instantiates two templates: (i) domain-specific templates that decompose a task into ordered stages with criticality scores, and (ii) a robust default template that follows a prefix-first prior for general queries. Under a token budget, we allocate large-model capacity to the most critical stages, or to a budgeted prefix under the default template.

Experiments show that VERTICAL ROUTING outperforms the strongest baseline (RouterDC) by 2.0 points in the averaged metric, while significantly reducing token usage by 48.9% and large-model output share by 41.7%. Additionally, it enhances stability by lowering the standard deviation by 87.5%. These results highlight its advantages in accuracy, efficiency, and robustness. [1]

1 Introduction

Large language models (LLMs) have recently shown strong performance across a wide range of tasks (Deng et al., 2025). But the size trade-off remains: large models handle hard problems better but are expensive, while small language models (SLMs) are cheaper but weaker (Šléher et al., 2025). This motivates routing strategies that reduce overall routing cost without sacrificing quality (Xue et al., 2023).

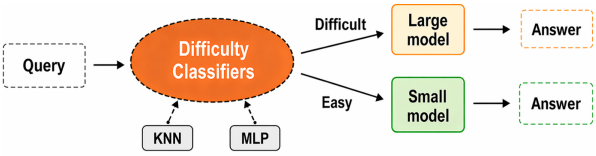
Current routing strategies typically infer task difficulty from the initial query and then assign the entire task to either an LLM or an SLM (Yu et al., 2025; Liu and Lo, 2025). We refer to this binary, query-level model selection paradigm as *Horizontal Routing*. As illustrated in Figure 1(a), this conventional pipeline makes a one-shot routing decision before generation begins. Figure 1(b) further shows that lightweight routers perform close to the 50% random baseline on balanced easy/hard sets across translation, math, and code (see Appendix A for details), suggesting that query-level signals alone are often insufficient for reliable routing. As a result, such methods are prone to misrouting and unstable behavior. Stronger evaluators, such as LLM-based pre-evaluation, can improve routing accuracy, but they also introduce substantial additional cost. Similar observations have been reported in recent studies, highlighting a central limitation of existing routing approaches (Kimi, 2025; Shen et al., 2025; Heitz et al., 2024).

The unreliability of difficulty prediction largely comes from treating a query as a single unit, ignoring heterogeneous sub-tasks within one response. We instead decompose generation into stages and assign each stage a *criticality score* that reflects its influence on the final outcome. Routing then becomes an intra-query allocation problem: prioritize the LLM on high-criticality stages and use the SLM for the rest to balance quality and cost.

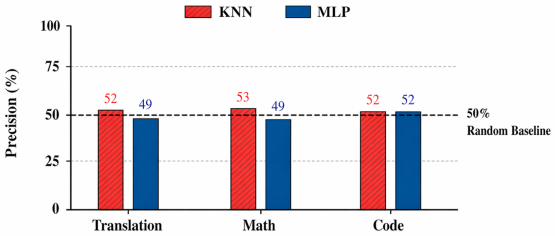
Defining stages depends on the task. For general queries, we adopt a positional prior: earlier tokens tend to have broader downstream impact and are harder to correct (Chen et al., 2024b), motivating a default prefix-first collaboration (LLM writes an initial prefix; SLM continues). For domain-specific queries, stages follow task structure; e.g., translation can first translate keywords

[1]The data and code are available at https://github.com/shenpeijun0212/Vertical-Routing.

Transactions of the Association for Computational Linguistics, vol. 14, pp. 2301–2317, 2026. https://doi.org/10.1162/TACL.a.802
Action Editor: Yuntian Deng. Submission batch: 8/2025; Revision batch: 4/2026; Published 9/2026.



(a) Horizontal Routing relies on one-shot query-level difficulty prediction.



(b) Lightweight query-level difficulty prediction is near random.

Figure 1: Lightweight routers remain close to the 50% random baseline on balanced easy/hard sets across translation, math, and code, indicating limited signal for query-level difficulty prediction.

and then generate the full sentence conditioned on these constraints (Wang et al., 2024), with similar decompositions in math (planning→solving) and other domains.

Informed by these considerations, we introduce VERTICAL ROUTING, a framework for collaborative inference between LLM and SLM. VERTICAL ROUTING has two components: *Template Definition* and *Template-based Routing*. We define (i) *domain-specific templates* that encode a small set of ordered stages with criticality scores based on domain knowledge, and (ii) a robust *default template* that follows a prefix-first collaboration pattern for general tasks and serves as a conservative fallback. At inference time, each dataset is paired with a routing template chosen once at the task level: when the task exhibits a consistent staged structure, a domain-specific template is used; otherwise, the default template serves as a fallback. It then executes a budget-aware routing policy: allocate the LLM to the most critical stages under budget, and apply a cost-driven greedy prefix allocation rule for the default template.

Experimental results demonstrate that VERTICAL ROUTING surpasses the strongest baseline, RouterDC. It achieves a 2.0-point gain in average performance (60.4 vs. 58.4) while simultaneously cutting token usage by 48.9% (474.5 vs. 929.1). Furthermore, it reduces the share of large-model outputs by 19.2 percentage points (from 46.0% to 26.8%), representing a relative reduction of 41.7%. Additionally, stability is markedly improved, evidenced by an 87.5% decrease in standard deviation. Collectively, these metrics underscore the method's superior accuracy, efficiency, and robustness.

This work offers three key contributions:

- **Stage-level routing.** We shift routing from a single task-level decision to allocating computation across stages within one query, guided by stage criticality.
- **Template-based collaboration.** We propose VERTICAL ROUTING with a set of routing templates, including domain-specific stage templates and a default template, enabling structured collaboration between LLM and SLM.
- **Better cost and stability.** Our evaluation show that VERTICAL ROUTING reduces cost and variance while maintaining comparable and slightly improved accuracy.

2 Methodology

2.1 Problem Formulation

We consider a dataset $\mathcal{D}$, where each instance is a query x. Given x, a routing method π coordinates a large model M_L and a small model M_S to produce a final output $\hat{y}(x;\pi)$. We evaluate the output with a task metric $s(\hat{y}, y^\star)$, such as accuracy or BLEU.

Cost. We define inference cost as the weighted token usage across all model calls. For each model $m \in \{L, S\}$, let T^m_{in} and T^m_{out} be the input and output token counts. In our collaborative routing framework, the small model processes both the original input x and the prefix y_L generated by M_L. Thus, the input lengths are defined as $T^L_{\text{in}} = \|x\|$ and $T^S_{\text{in}} = \|[x; y_L]\|$, respectively. Given a per-token cost coefficient $c_m > 0$, the total cost is formulated as:

$$C(x;\pi) = \sum_{m\in\{L,S\}} c_m \Big(T^m_{\text{in}} + T^m_{\text{out}}\Big). \quad (1)$$

The coefficients c_m can be obtained via explicit pricing for closed-source models or estimated via a micro-benchmark for open-source models. For the latter, we measure average decoding latency (ms/token) and set the relative cost ratio $r \triangleq c_L/c_S \approx \text{ms/token}(M_L)/\text{ms/token}(M_S)$. We then use $c_S = 1$ and $c_L = r$ to obtain a reproducible relative cost.

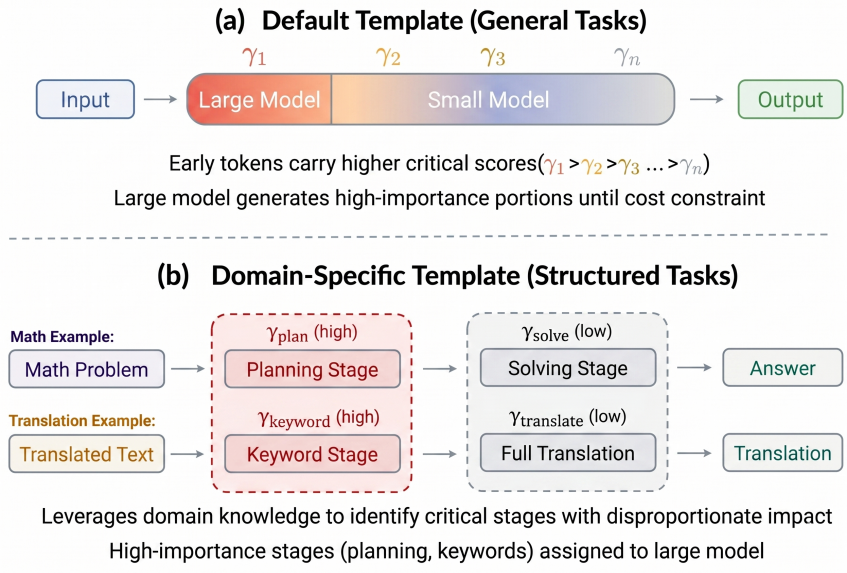


Figure 2: Overview of VERTICAL ROUTING. A routing template is selected once per task family and then executed for each query. Under domain-specific templates, the LLM handles high-criticality stages (e.g., solution planning or keyword identification), while the SLM completes the remaining stages. Under the default prefix-first template, the LLM generates a budgeted high-criticality prefix and the SLM continues the remainder.

Objectives. Our primary goal is to maximize the expected task performance under a budget $\mathcal{B}$:

$$\begin{aligned} &\max_{\pi} \ \mathbb{E}\left[S^{(k)}\right], \\ &\text{s.t. } \mathbb{E}_{x\sim\mathcal{D}}[C(x;\pi)] \le \mathcal{B}, \end{aligned} \tag{2}$$

where $S^{(k)}$ is the average score over dataset $\mathcal{D}$ for run k. We treat stability as a secondary objective. Since complex generation is often sensitive to decoding randomness, we prefer routing policies that exhibit smaller run-to-run variance $\mathrm{Var}_k\left[S^{(k)}\right]$ to ensure reliable deployment.

2.2 Method Overview

Inspired by prior work on cost-efficient multi-model inference and cascaded generation (Leviathan et al., 2023; Dekoninck et al., 2025), VERTICAL ROUTING, shown in Figure 2, reframes routing from query-level model selection to intra-query stage allocation. Instead of assigning an entire query to either a large or small model, it decomposes generation into ordered stages and reserves the large-model budget for the most critical ones.

The framework has two phases:

- **Task-level Template Selection:** For each task family (a dataset), the user selects a routing template once based on the task description and a small subset. A domain-specific template is used only when instances share the same ordered stage decomposition, later stages depend on earlier-stage outputs, and stage criticality differs clearly. Otherwise, the default prefix-first template is used.

- **Budget-Induced Stage Allocation:** Given a budget $\mathcal{B}$, the framework computes a large-model token capacity B and greedily assigns the large model M_L to the most critical stages according to criticality scores γ_i, while the small model M_S completes the remaining stages.

2.3 Template Definition

A template T defines the logical decomposition of a query's response into a sequence of stages. Formally, a template is a set of ordered pairs: $T = \{(s_1, \gamma_1), \ldots, (s_n, \gamma_n)\}$, where each stage s_i is associated with a *criticality score* $\gamma_i \in \mathbb{R}^+$.

2.3.1 Domain-Specific Templates: Structural Prior

Inspired by the observation that specialized tasks exhibit a non-uniform internal structure, decision-making stages often exert a critical influence on final result relative to execution (Zhou et al., 2023; Khot et al., 2023), we propose domain-specific templates to concentrate expensive computation on these high-criticality bottlenecks.

Definition. A domain template T_{domain} identifies key stages based on expert knowledge (e.g., *Plan-and-Solve* for mathematics or *Keyword-Constraint* for translation). We assign higher γ_i to **decision-making stages** that impose hard constraints on the output space, and lower γ_i to **execution stages** that follow those constraints.

2.3.2 Default Template: Positional Prior

The rationale for the default template is grounded in the positional prior of autoregressive generation (Ranzato et al., 2016), where early tokens serve as the foundation for all subsequent steps. Because initial deviations can compound and lead to significant semantic drift through exposure bias, the quality of the prefix is disproportionately vital to the final output. We empirically validate this prior in Appendix B through budget-matched experiments, which demonstrate that generating the prefix with the large model consistently outperforms the reverse sequence under identical token constraints.

Definition. Building on this insight, we propose the *prefix-first* default template T_{default} for general tasks where internal structural importance is latent. This template follows a sequential decomposition that assigns the highest criticality to the starting tokens of a response. Under this framework, the large model M_L is prioritized to establish a high-quality foundational prefix, after which the small model M_S continues the decoding process until completion.

2.4 Template-based Routing Strategy

2.4.1 Template Selection

Rather than relying on brittle per-query difficulty estimation, we assign a routing template once for task family (a dataset in our benchmark setup):

$$T_{\mathcal{D}} \leftarrow g(\tau(\mathcal{D})) \in \mathcal{T}.$$

Here, $\tau(\mathcal{D})$ denotes the reusable response structure shared by most instances in $\mathcal{D}$, and $g(\cdot)$ maps that structure to a routing template. Template selection is therefore a coarse-grained task-level decision rather than a per-query routing step: once a template is chosen for a task family, all queries in that family follow the same template.

In practice, the user first reads the task description and then inspects

$$n_{\text{check}} = \min(20, \max(10, \lceil 0.05\,|\mathcal{D}_{\text{cal}}| \rceil)),$$

representative examples from a subset $\mathcal{D}_{\text{cal}}$, or all available examples when fewer than 10 are available. We use a domain-specific template only when all three checks below are satisfied:

1. **Shared stage structure:** Can most instances be decomposed with the same ordered stages?

2. **Stage dependency:** Does the later stage explicitly rely on the output of the earlier stage?

3. **Stage criticality gap:** Do the stages differ clearly in criticality?

This procedure is intended as a lightweight, conservative checklist applied once per task family. The first two checks are usually clear from the task description and a small representative subset, while the third may require limited practitioner judgment. In deployment, when manual selection is undesirable, task-level screening could in principle be assisted by a lightweight model or embedding-based signals at minimal additional cost. We do not study such automation here. Whenever the evidence is ambiguous, we fall back to the default prefix-first template. This keeps template assignment low-cost and low-risk. Concrete examples and full specialized templates are provided in Appendix C.

2.4.2 Budget-to-Capacity Mapping

A practical challenge in routing is that financial budgets $\mathcal{B}$ cannot be enforced directly during autoregressive decoding; instead, the system can only regulate the number of generated tokens via the `max_new_tokens` parameter. To bridge this gap, we map the budget $\mathcal{B}$ into an allowable **output-token capacity** B for the large model M_L.

We define $K_{\max}$ as the planned total output length. When global statistics are unavailable, we estimate K_{max} via held-out calibration subset, calculating the average generation length on a small subset to serve as the expected budget baseline. As shown in Eq. (1), the total cost comprises a baseline expenditure (assuming all $K_{\max}$ tokens are generated by M_S) and a premium cost for upgrading selected tokens to M_L. When global statistics are unavailable, we estimate $K_{\max}$ from deployment statistics or a held-out calibration subset, rather than from the evaluation set.

2.4.3 Stage Allocation for Domain-Specific Templates

For domain templates, let ℓ_i be the planned token quota for stage s_i and $x_i \in \{0, 1\}$ be the indicator for M_L assignment. Under budget $\mathcal{B}$, we solve for the assignment vector $\mathbf{x}$ subject to:

$$c_S K_{\max} + (c_L - c_S) \sum_{i=1}^{m} x_i \ell_i \leq \mathcal{B}, \quad (3)$$

where $c_L - c_S$ represents the marginal cost of using M_L over M_S. This yields the total large-model capacity B:

$$B = \min \left\{ K_{\max}, \max \left(\frac{\mathcal{B} - c_S K_{\max}}{c_L - c_S}, 0 \right) \right\}. \quad (4)$$

In our proxy calculation, we assume $\mathcal{B}$ is the remaining budget after subtracting the fixed input costs, as input lengths T_{in}^L and T_{in}^S are determined before output generation begins.

Greedy Selection. We prioritize allocating M_L to the most critical stages by sorting stages in descending order of their criticality density γ_i/ℓ_i. We set $x_i = 1$ for the top-ranked stages until the cumulative length $\sum x_i \ell_i$ reaches B, after which the remaining stages are handled by M_S.

2.4.4 Allocation for Default Templates

For T_{default}, we apply a *prefix-first* rule. Using B calculated from Eq. (4), M_L is assigned to generate the initial B tokens of the response, after which M_S continues decoding until completion.

3 Experiments

3.1 Setup

Datasets. We evaluate on four benchmarks spanning reasoning and generation: GSM8K (Cobbe et al., 2021) and MATH (Hendrycks et al., 2021) for mathematical reasoning, WMT (Kocmi et al., 2022) for machine translation, and MBPP (Austin et al., 2021) for code generation with unit tests.

Models. We use pre-trained models from the Qwen2.5 family (Qwen et al., 2025) as the large model M_L and the small model M_S, which provides strong models at multiple scales for budget-controlled comparison.

Baselines. We compare VERTICAL ROUTING with KNN-Router and MLP-Router (Hu et al., 2024), RouteLLM (Ong et al., 2025), RouterDC (Chen et al., 2024a), and FrugalGPT (Chen et al., 2023). For all baselines, we use the best settings reported in the original papers or official implementations; details are provided in Appendix D.

Template configuration. Following Sec. 2.4.1, templates are selected once per task family using only non-test information. We use the Plan-and-Solve (PS) template for GSM8K and MATH, the Deep Reasoning Translation (DRT) template for WMT, and the default prefix-first template for MBPP. Appendix C provides the full specialized templates, including DRT. Appendix E.1 additionally compares prefix-first with PS on GSM8K. We do not tune the prefix budget on the evaluation set; any such calibration uses held-out data only.

Cost and evaluation. We measure compute by total token usage, i.e., the sum of input and output tokens across all model calls, including routing or pre-evaluation overhead. We also report the large-model output share

$$\rho_L = 100 \times \frac{\mathbb{E}[T_{\text{out}}^L]}{\mathbb{E}[T_{\text{out}}^L + T_{\text{out}}^S]},$$

where T_{out}^L and T_{out}^S are the output tokens produced by M_L and M_S, respectively.

For domain-specific templates, M_L generates the key stage and M_S completes the remaining stages, so ρ_L is computed from the realized stage-level outputs. For the default prefix-first template on MBPP, we compare methods under a matched expected large-model budget B. Since binary baselines route each query to only one model, we enforce the same expected budget by limiting their large-model usage frequency: $\pi_L = B/\mu_L$, where μ_L is the average generation length of the large model, and we route the top-π_L fraction of queries to M_L.

For transparency, Table 1 reports raw total token counts across all model invocations, including repeated input processing when both models are used. These counts are unweighted by model size. Because per-token cost differs across 3B and 32B models, we additionally report a simple size-weighted cost proxy in Appendix F.

We use accuracy for GSM8K, MATH, and MBPP, and BLEU-4 (Papineni et al., 2002) for WMT. All metrics are linearly rescaled to $[0, 100]$ for unified comparison. Experiments were conducted on 4 NVIDIA A800-SXM4-80GB GPUs.

Table 1: Results on GSM8K, MATH, and WMT. We report task performance (Acc for GSM8K/MATH, BLEU for WMT), token usage (Tok.; raw, unweighted total input+output tokens over all model calls, including routing overhead), and the large-model output share ρ_L (in %), where $\rho_L = 100 \times \mathbb{E}[T^l_{out}]/\mathbb{E}[T^l_{out} + T^s_{out}]$. **Perf.** is the arithmetic mean of the three rescaled task scores (all metrics linearly mapped to $[0, 100]$). The best is in bold and the second-best is underlined.

	GSM8K			MATH			WMT			AVG		
	Acc	Tok.	ρ_L	Acc	Tok.	ρ_L	BLEU	Tok.	ρ_L	Perf.	Tok.	ρ_L
Qwen2.5-3B	75.0	211.6	-	38.6	301.2	-	24.4	410.3	-	46.0	307.7	-
Qwen2.5-32B	90.8	989.3	-	57.3	1647.9	-	41.5	1380.2	-	63.2	1339.1	-
KNN-Router	77.3	**379.2**	**21.3**	45.6	580.3	39.8	32.4	605.3	20.1	51.8	521.6	27.1
MLP-Router	77.9	418.9	26.7	46.2	620.7	40.3	32.9	610.5	20.6	52.3	550.0	29.2
RouteLLM	81.4	785.4	36.3	48.9	790.5	47.6	35.7	752.1	31.2	55.3	776.0	38.4
RouterDC	85.2	881.5	46.4	52.7	925.4	52.9	37.2	980.4	38.8	58.4	929.1	46.0
FrugalGPT	82.7	801.6	44.9	53.1	905.8	53.0	34.6	690.8	27.6	56.8	799.4	41.8
Ours	**87.8**	407.4	23.7	**54.5**	**490.9**	**38.4**	**38.8**	**525.1**	**18.4**	**60.4**	**474.5**	**26.8**

3.2 Main Results

Table 1 reports performance, total token usage (Tok.), and large-model output share (ρ_L) on GSM8K, MATH, and WMT. Our method achieves the best performance across all three tasks (87.8 on GSM8K, 54.5 on MATH, and 38.8 BLEU on WMT) while maintaining low inference costs. Compared to strong routing baselines like RouteLLM, RouterDC, and FrugalGPT, our token usage is consistently lower. For example, on GSM8K, we use only 407.4 tokens, whereas the baselines require 785.4, 881.5, and 801.6. We observe similar margins on MATH (490.9 vs. 790.5, 925.4, and 905.8) and WMT (525.1 vs. 752.1, 980.4, and 690.8). On average, we obtain the highest performance (60.4) with the lowest token consumption (474.5). Our average ρ_L is 26.8%, notably lower than RouteLLM (38.4%), RouterDC (46.0%), and FrugalGPT (41.8%), indicating that we attain stronger results with less reliance on large-model outputs.

Figure 3 illustrates MBPP accuracy as a function of the large-model quota π_L. Our method is particularly effective in the low-budget regime. At $\pi_L = 10\%$, we achieve an accuracy of 54.5, surpassing RouterDC (51.0), FrugalGPT (50.5), and RouteLLM (48.0). This advantage persists as the quota increases. At $\pi_L = 20\%$ and 40%, our accuracy rises to 58.0 and 60.5 respectively, maintaining a clear lead over all baselines. While performance improves monotonically with π_L and eventually approaches the large-model upper bound, the gains are most significant at lower budgets. These results confirm that our approach utilizes the limited budget more efficiently than competing methods.

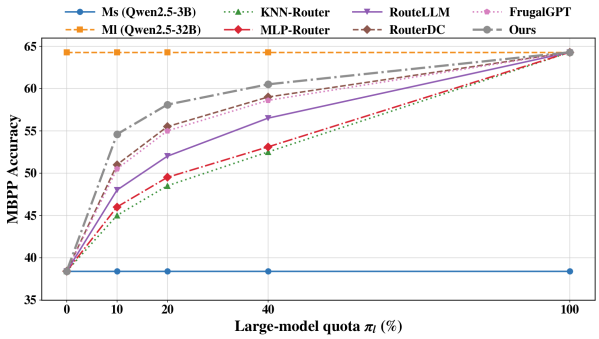


Figure 3: MBPP accuracy vs. large-model quota π_L. Queries are routed to M_l based on the top-π_L router scores.

3.3 Stability Analysis

Beyond average performance, routing methods should be *stable* under repeated runs, since complex generation can be sensitive to decoding randomness. We evaluate stability by repeating each method ten times on each dataset under the same decoding configuration and reporting the standard deviation of the task metric across runs.

As shown in Figure 4, Vertical Routing is consistently stable across all four datasets, with standard deviation below 0.4%. Averaged over datasets, it reduces standard deviation by 91.87% (KNN-Router), 91.07% (MLP-Router), 89.47% (RouteLLM), 87.50% (RouterDC), and 87.95% (FrugalGPT).

Compared with difficulty-based routers (KNN-Router, MLP-Router, and RouteLLM), which can

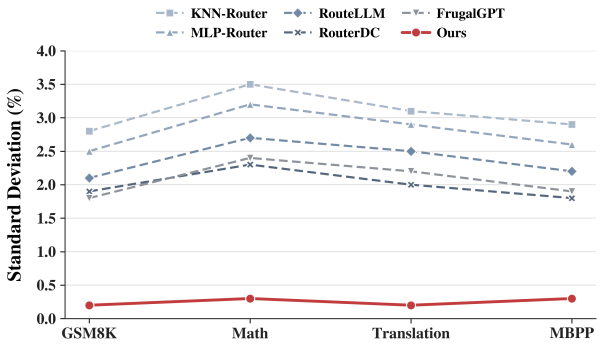


Figure 4: Stability comparison across ten runs. Error bars indicate standard deviation of the task metric.

fluctuate due to routing errors and sensitivity to query-level predictions, Vertical Routing is more stable because it follows a deterministic, template-driven collaboration policy once a template is selected. Relative to multi-stage systems such as RouterDC and FrugalGPT, our method also shows lower variance, suggesting that budget-driven stage allocation leads to more consistent behavior across runs.

3.4 Cost-Performance Analysis

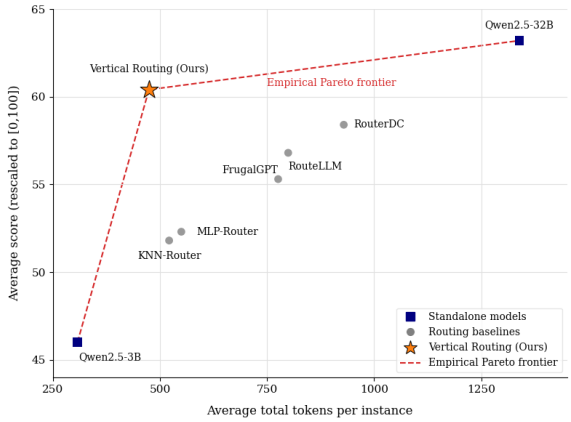


Figure 5: Cost–performance trade-off of standalone models and routing methods. Each point shows average performance versus total token usage. VERTICAL ROUTING lies on the empirical Pareto frontier together with the two standalone endpoints.

As shown in Figure 5, VERTICAL ROUTING lies on the empirical Pareto frontier together with the two standalone endpoints, Qwen2.5-3B and Qwen2.5-32B. Among all routing methods, it is the only nondominated point: it achieves the highest average performance (60.4) while using fewer total tokens (474.5) than every routing baseline. All competing routing methods lie below this frontier, indicating less favorable cost–performance trade-offs.

Compared with the strongest baseline, RouterDC, VERTICAL ROUTING improves average performance by 2.0 points (60.4 vs. 58.4) while reducing token usage by 48.9% (474.5 vs. 929.1). This pattern is consistent with the design of our framework. Instead of relying on query-level difficulty prediction or assigning an entire query to a single model, VERTICAL ROUTING reserves the large model for critical stages, or for a budgeted prefix under the default template, and lets the small model complete the remaining generation. Its advantage therefore comes from more effective large-model budget allocation, rather than greater large-model usage.

4 Case Study: Error Analysis

To characterize the limitations of VERTICAL ROUTING, we analyze failure instances under the operating points from Table 1. Leveraging the explicit nature of intermediate outputs (plans/keywords), we trace errors to three structural sources: the **LLM-Critical Stage**, the **SLM-Completion Stage**, or (for MBPP) **Quota Allocation**.

4.1 Error Taxonomy & Attribution

We categorize errors based on the earliest point of failure:

- **Critical-Stage Failure (M_L):** The plan (PS) or keywords (DRT) contain incorrect assumptions or translations. Since M_S is conditioned on this output, the error propagates inevitably.
- **Completion-Stage Failure (M_S):** The critical output is correct, but M_S fails during execution. This includes *Execution Errors* (arithmetic slips), *Constraint Violations* (ignoring keywords), or *Handoff Failures* (deviating from the plan).
- **Allocation Failure (Router):** Specifically for MBPP, complex instances are not selected into the top-π_L quota, forcing M_S to handle tasks beyond its capability.

4.2 Key Findings

1. Propagation of Critical Errors. The system's correctness is upper-bounded by M_L. When

Table 2: Representative error tracing.

Type	Description & Diagnosis
Exec. Error	**Plan** (M_L)**:** Correctly sets up equations. **Comp** (M_S)**:** Arithmetic slip in subtraction. → *SLM capacity limitation.*
Plan Error	**Plan** (M_L)**:** Uses wrong formula for compound interest. **Comp** (M_S)**:** Solves correctly based on wrong formula. → *Error propagation from* M_L.
Constraint	**Key** (M_L)**:** Correct terminology provided. **Comp** (M_S)**:** Paraphrases term, violating constraint. → *Handoff failure (SLM instruction following).*

M_L generates a flawed plan or mistranslates a keyword, M_S often executes this flaw faithfully. This "blind obedience" confirms that the small model relies heavily on the critical stage for guidance but lacks the mechanism to correct upstream errors.

2. The Execution Bottleneck. Even with perfect planning, M_S constitutes a reliability bottleneck. In Math, we observe frequent arithmetic errors despite correct formulas; in Translation, M_S occasionally drops constraints. This highlights a trade-off: vertical collaboration reduces cost but exposes the system to the inherent limitations of the small model's context adherence and reasoning precision.

3. Allocation Sensitivity. For MBPP, a significant portion of failures stems from the router's inability to prioritize the "hardest" prompts for the limited M_L budget. This suggests that improving the ranking signal is as critical as improving the generation models themselves.

4.3 Representative Cases

Table 2 summarizes typical failure modes.

Implications. Our analysis points to future directions: strengthening critical-stage verification to stop error propagation, and improving M_S's instruction following to ensure smoother handoffs.

5 Related Works

5.1 Traditional LLM Routing Approaches

Early research on model routing used methods like external classifiers (da Rocha, 2022) or heuristic rules, as in Mixture-of-Experts (Shazeer et al., 2017), to predict query complexity. However, these approaches often suffered from poor generalization, as superficial features proved to be unreliable proxies for true task difficulty. More recent work leverages the LLM's own uncertainty estimates for scoring (Kadavath et al., 2022), but this self-referential method can introduce dependency loops that diminish cost savings. At a finer grain, while CITER (Zheng et al., 2025) uses token-level routing for collaborative inference, its per-token decisions risk significant computational overhead and instability. Our routing framework shifts the focus from predicting the overall difficulty of tasks to allocating the importance of each stage of internal sub-tasks.

5.2 Cost-Based LLM Cascading

Cascading strategies offer a parallel line of research that explores dynamic model cascades (Yue et al., 2024), which switch from smaller, faster models to larger, more capable ones based on intermediate confidence scores or partial outputs, a concept with parallels in computer vision (Wang et al., 2021). However, while effective for routing simple queries, these methods are often hampered by the need for brittle, hand-tuned confidence thresholds that do not generalize well across datasets. They also incur significant sunk computational costs when the output of a smaller model is discarded upon escalation to a larger one (Jame et al., 2022). To refine the routing signal, Gupta et al. (2024) introduce token-level uncertainty measures that correct for the length bias inherent in sequence-level confidence scores. Shifting focus from inference confidence to system-level metrics, Oh et al. (2024) develop ExeGPT for constraint-aware resource scheduling, which optimizes for objectives. In contrast to these approaches, our VERTICAL ROUTING effectively balances efficiency and cost through negligible task classification and flexible template routing strategies, avoiding the pitfalls of threshold-based escalation.

5.3 Multi-Model Collaboration

Research into multi-model collaboration has explored system-level optimizations, such as PerLLM by Yang et al. (2024), which uses personalized scheduling for edge-cloud inference based on network latency and device capabilities. However, its focus is on the logistical orchestration of where and when to execute complete models, treating them as monolithic tasks rather than decomposing the cognitive workload. Other advanced routing approaches like RouterDC employ contrastive learning to navigate the quality-cost trade-off, while SplitReason (Akhauri et al., 2025) uses reinforcement learning for alternating reasoning between models. While these methods represent progress in intelligent model selection, they still rely on complete task assignments rather than a more granular cognitive decomposition. In contrast, we assign collaboration between large and small models based on the critical score of each reasoning stage, enabling a true division of cognitive labor.

6 Conclusion

We introduced VERTICAL ROUTING, a collaborative inference framework that replaces query-level difficulty prediction with dataset-level template selection and stage-level budget allocation. By assigning the large model to critical stages (or a budgeted prefix under the default template) and letting the small model complete the remaining generation, VERTICAL ROUTING enables a simple and robust division of labor within a single answer.

Empirically, across GSM8K, MATH, and WMT, VERTICAL ROUTING achieves the best average performance while using substantially fewer tokens and a smaller large-model output share than strong routing baselines. On MBPP, under matched expected large-model budgets enforced by a routing quota, it remains most effective in the low-budget regime. Moreover, across repeated runs, VERTICAL ROUTING exhibits markedly lower variance than query-level routing methods, highlighting its robustness for deployment.

Future work includes improving the reliability of critical-stage outputs (to reduce error propagation), strengthening completion-stage execution (e.g., lightweight verification for arithmetic and constraint satisfaction), and extending domain templates beyond math and translation.

7 Limitations

VERTICAL ROUTING is most effective for recurring task families, where task-family identification, template selection, and prefix-budget calibration can be reused across many instances. These are task-level setup costs rather than per-query costs. The framework does not require synchronous large-batch inference, but its advantages are less well amortized in smaller, highly heterogeneous, or long-tailed workloads, especially when many task families have only a small number of examples. Specialized templates are optional rather than universal: when task structure is unclear, the default prefix-first template serves as a conservative fallback. We also do not systematically evaluate automatic grouping, classification noise, or very small-batch settings.

The framework is further limited by the execution ability of the small model. Even with strong guidance from the large model, failures can persist when the small model must maintain long-range dependencies or strict execution correctness, as in coding tasks.

Our experiments use a single model family for controlled comparison. While the framework itself is defined at the level of text-based stage handoff and is not restricted to same-family pairings, cross-family combinations may require interface normalization and separate calibration.

Acknowledgments

We thank the TACL Action Editor and anonymous reviewers for their constructive feedback. This research was supported by the Key Program of the National Social Science Fund of China (Grant No. 24ATQ009).

References

Yash Akhauri, Anthony Fei, Chi-Chih Chang, Ahmed F. AbouElhamayed, Yueying Li, and Mohamed S. Abdelfattah. 2025. Splitreason: Learning to offload reasoning.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models.

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

Shuhao Chen, Weisen Jiang, Baijiong Lin, James Kwok, and Yu Zhang. 2024a. Routerdc: Query-based router by dual contrastive learning for assembling large language models. *Advances in Neural Information Processing Systems*, 37:66305–66328.

Xinyun Chen, Ryan A. Chi, Xuezhi Wang, and Denny Zhou. 2024b. Premise order matters in reasoning with large language models.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems.

Jasper Dekoninck, Maximilian Baader, and Martin Vechev. 2025. A unified approach to routing and cascading for LLMs. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 12987–13010. PMLR.

Hexuan Deng, Wenxiang Jiao, Xuebo Liu, Jing Li, Min Zhang, and Zhaopeng Tu. 2025. Drpruning: Efficient large language model pruning through distributionally robust optimization.

Neha Gupta, Harikrishna Narasimhan, Wittawat Jitkrittum, Ankit Singh Rawat, Aditya Krishna Menon, and Sanjiv Kumar. 2024. Language model cascades: Token-level uncertainty and beyond.

Luc Bryan Heitz, Joun Chamas, and Christopher Scherb. 2024. Evaluation of the programming skills of large language models.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring massive multitask language understanding.

Qitian Jason Hu, Jacob Bieker, Xiuyu Li, Nan Jiang, Benjamin Keigwin, Gaurav Ranganath, Kurt Keutzer, and Shriyash Kaustubh Upadhyay. 2024. Routerbench: A benchmark for multi-llm routing system.

Ahmadreza Jame, Marco Rapelli, and Claudio Casetti. 2022. Reducing pollutant emissions through virtual traffic lights. *Computer Communications*, 188:167–177.

Saurav Kadavath, Tom Conerly, Amanda Askell, Tom Henighan, Dawn Drain, Ethan Perez, Nicholas Schiefer, Zac Hatfield-Dodds, Nova DasSarma, Eli Tran-Johnson, Scott Johnston, Sheer El-Showk, Andy Jones, Nelson Elhage, Tristan Hume, Anna Chen, Yuntao Bai, Sam Bowman, Stanislav Fort, Deep Ganguli, Danny Hernandez, Josh Jacobson, Jackson Kernion, Shauna Kravec, Liane Lovitt, Kamal Ndousse, Catherine Olsson, Sam Ringer, Dario Amodei, Tom Brown, Jack Clark, Nicholas Joseph, Ben Mann, Sam McCandlish, Chris Olah, and Jared Kaplan. 2022. Language models (mostly) know what they know.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2023. Decomposed prompting: A modular approach for solving complex tasks. In *International Conference on Learning Representations (ICLR)*.

Team Kimi. 2025. Kimi k1.5: Scaling reinforcement learning with llms.

Tom Kocmi, Rachel Bawden, Ondřej Bojar, Anton Dvorkovich, Christian Federmann, Mark Fishel, Thamme Gowda, Yvette Graham, Roman Grundkiewicz, Barry Haddow, Rebecca Knowles, Philipp Koehn, Christof Monz, Makoto Morishita, Masaaki Nagata, Toshiaki Nakazawa, Michal Novák, Martin Popel, and Maja Popović. 2022. Findings of the 2022 conference on machine translation (WMT22). In *Proceedings of the Seventh Conference on Machine Translation (WMT)*, pages 1–45, Abu Dhabi, United Arab Emirates (Hybrid). Association for Computational Linguistics.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 19274–19286. PMLR.

Kuan-Ming Liu and Ming-Chih Lo. 2025. Llm-

based routing in mixture of experts: A novel framework for trading.

Hyungjun Oh, Kihong Kim, Jaemin Kim, Sungkyun Kim, Junyeol Lee, Du seong Chang, and Jiwon Seo. 2024. ExeGPT: Constraint-aware resource scheduling for LLM inference.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M Waleed Kadous, and Ion Stoica. 2025. Routellm: Learning to route llms with preference data.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 311–318, Philadelphia, Pennsylvania, USA. Association for Computational Linguistics.

Qwen, 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, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. 2025. Qwen2.5 technical report.

Marc'Aurelio Ranzato, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2016. Sequence level training with recurrent neural networks. In *International Conference on Learning Representations (ICLR)*.

Roldao da Rocha. 2022. Holographic entanglement entropy, deformed black branes, and deconfinement in ads/qcd. *Physical Review D*, 105(2).

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

Yi Shen, Jian Zhang, Jieyun Huang, Shuming Shi, Wenjing Zhang, Jiangze Yan, Ning Wang, Kai Wang, Zhaoxiang Liu, and Shiguo Lian. 2025. Dast: Difficulty-adaptive slow-thinking for large reasoning models.

Jiaan Wang, Fandong Meng, Yunlong Liang, and Jie Zhou. 2024. Drt-o1: Optimized deep reasoning translation via long chain-of-thought. *arXiv e-prints*, pages arXiv–2412.

Yulin Wang, Rui Huang, Shiji Song, Zeyi Huang, and Gao Huang. 2021. Not all images are worth 16x16 words: Dynamic transformers for efficient image recognition.

Fuzhao Xue, Valerii Likhosherstov, Anurag Arnab, Neil Houlsby, Mostafa Dehghani, and Yang You. 2023. Adaptive computation with elastic input sequence. In *International Conference on Machine Learning*, pages 38971–38988. PMLR.

Zheming Yang, Yuanhao Yang, Chang Zhao, Qi Guo, Wenkai He, and Wen Ji. 2024. PerLLM: Personalized inference scheduling with edge-cloud collaboration for diverse LLM services.

Qianjin Yu, Keyu Wu, Zihan Chen, Chushu Zhang, Manlin Mei, Lingjun Huang, Fang Tan, Yongsheng Du, Kunlin Liu, and Yurui Zhu. 2025. Rethinking the generation of high-quality cot data from the perspective of llm-adaptive question difficulty grading.

Murong Yue, Jie Zhao, Min Zhang, Liang Du, and Ziyu Yao. 2024. Large language model cascades with mixture of thoughts representations for cost-efficient reasoning.

Wenhao Zheng, Yixiao Chen, Weitong Zhang, Souvik Kundu, Yun Li, Zhengzhong Liu, Eric P. Xing, Hongyi Wang, and Huaxiu Yao. 2025. CITER: Collaborative inference for efficient large language model decoding with token-level routing.

Denlu Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Carigut, William Fedus, V Le Quoc, and Denny Zhou. 2023. Least-to-most prompting enables complex reasoning in large language models. In *International Conference on Learning Representations (ICLR)*.

Richard Šléher, William Brach, Tibor Sloboda, Kristián Košťál, and Lukas Galke. 2025. Guarded query routing for large language models.

A Lightweight Routers Are Close to Random Guess

This appendix evaluates whether lightweight routers can reliably predict query difficulty from the initial query alone. We focus on KNN-Router and MLP-Router, and test their ability to decide whether an input should be routed to the LLM or the SLM.

Constructing easy vs. hard instances. For each domain (translation, math, and code), we construct a binary difficulty set by forming two groups: *easy* instances that the small model can handle well, and *hard* instances that require the large model. For math and code, we label an instance as *easy* if the SLM produces a correct solution under the same decoding setup, and as *hard* if the SLM fails but the LLM succeeds. For translation, we follow the same principle by using sentence-level quality: an instance is *easy* if the SLM achieves high translation quality, and *hard* if the SLM quality is low while the LLM is substantially better. We then subsample to make the two groups balanced (50/50), so random guessing yields 50% accuracy. Given a query as input, the router predicts whether it belongs to the *hard* group (route to LLM) or the *easy* group (route to SLM).

Results. As shown in Figure 1, both routers perform close to the 50% random baseline across domains. KNN-Router achieves about 51.8% (Translation), 52.8% (Math), and 51.8% (Code), while MLP-Router achieves about 48.9%, 48.9%, and 51.8%, respectively. These results indicate that query-level difficulty prediction using lightweight routers provides limited signal and can be unreliable, motivating routing strategies that do not hinge on a single up-front difficulty decision.

B Budget-Matched Order Comparison for the Default Template

Under the same large-model token budget, letting the large model generate the prefix (L→S) is consistently better than letting the small model generate the prefix (S→L).

Setup. We evaluate on the GSM8K test set with 1,319 instances, using Qwen2.5-32B as the large model M_l and Qwen2.5-3B as the small model M_s. We fix the decoding configuration and the total generation cap K (`max_new_tokens`) for the final answer. Given a cost budget, we compute the large-model output budget via Eq. (4), which yields a large-model token capacity B (equivalently, a fraction $p = B/K$).

Under the *same* large-model token budget B, we compare two continuation orders: L→S lets M_l generate up to $\min(B, K)$ tokens, then M_s continues until reaching K; S→L lets M_s generate the prefix, and M_l generates the remaining tokens up to the same budget B. Thus, the two settings differ only in the *order* of model usage, not in the amount of large-model generation.

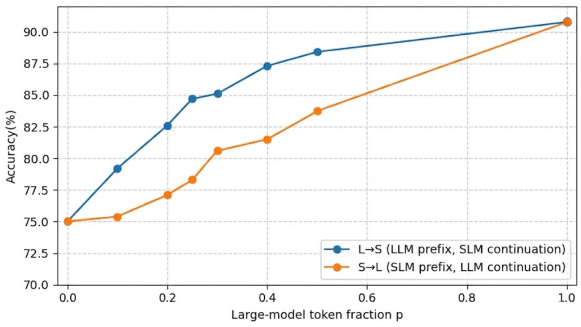


Figure 6: Budget-matched comparison on GSM8K (Qwen2.5-32B/3B). For the same large-model token fraction p, L→S consistently outperforms S→L.

Results. Figure 6 shows that L→S outperforms S→L across a wide range of budgets. When $p \in [0.1, 0.5]$, L→S achieves higher accuracy by about 4–6 points (e.g., 79.2 vs. 75.4 at $p = 0.1$, 84.7 vs. 78.3 at $p = 0.25$, and 88.4 vs. 83.7 at $p = 0.5$). As expected, the two orders coincide at the endpoints: at $p = 0$ both reduce to the small model (75.0%), and at $p = 1$ both reduce to the large model (90.6% vs. 90.2%). These results support the prefix-first prior used by the default template: with the same large-model budget, allocating the large model to early generation yields better overall outcomes than allocating it to later tokens.

C Template Selection and Prompt Templates

Once a task family is deemed suitable for a domain-specific template, we design the template by identifying a minimal reusable stage decomposition, requiring an explicit intermediate handoff between stages, and assigning the large model to the stage whose errors are most propagation-prone.

C.1 Applying the Template-Selection Checklist: GSM8K and MBPP

To make the template-selection procedure concrete, we apply the three checks from Sec. 2.4.1 to two representative task families: GSM8K, which is assigned a domain-specific template, and MBPP, which uses the default prefix-first template. In both cases, the decision is made once at the task level after reading the task description and inspecting a small non-test development/calibration subset; the template is not reselected for each individual query.

GSM8K → Plan-and-Solve (PS). For GSM8K, all three checks are satisfied.

- **Check 1 (shared stage structure):** Most GSM8K problems follow a reusable *plan → solve* structure. A first stage identifies the relevant quantities and the operation sequence, and a second stage executes that plan to reach the final answer.
- **Check 2 (stage dependency):** The later stage explicitly depends on the earlier stage. The solution stage uses the plan produced in the first stage as its scaffold for computation.
- **Check 3 (stage criticality gap):** The planning stage is more critical. If the wrong quantities or operation order are selected, the later execution stage will usually follow that mistake, producing a hard-to-recover error. By contrast, the second stage is mainly execution of the identified plan.

Since all three checks are satisfied, GSM8K is assigned the domain-specific PS template.

MBPP → Default prefix-first. For MBPP, the evidence for a reusable domain-specific template is weaker, so we use the default prefix-first template.

- **Check 1 (shared stage structure):** Although many MBPP instances can loosely be viewed as *understand the specification → write the code*, the intermediate structure is not stable enough to define a single reusable task-specific handoff template across most problems. Different instances require different kinds of decomposition, such as arithmetic logic, string processing, recursion, or data-structure manipulation.
- **Check 2 (stage dependency):** The final code depends on the full problem statement and unit tests, not simply on executing a previously generated intermediate artifact. A short plan can help, but it is not a sufficiently explicit and reliable scaffold in the same way as a math plan or a terminology list in translation.
- **Check 3 (stage criticality gap):** Criticality is less cleanly separated across stages. In code generation, correctness can fail at many points in the program, and later tokens may either introduce or repair bugs. Thus, we do not observe a single reusable stage whose errors are consistently more propagation-prone across most instances.

Because the checklist does not provide strong evidence for a specialized template, MBPP falls back to the default prefix-first template. This conservative fallback still lets the large model provide a strong early scaffold, without imposing a brittle task-specific decomposition.

These two examples illustrate the intended use of the checklist: we adopt a domain-specific template only when shared stage structure, explicit stage dependency, and non-uniform stage criticality are all clearly supported at the task level; otherwise, we conservatively use the default template.

Template–task alignment. In our framework, a template is considered aligned with a task family when three properties recur across representative instances: the same ordered stage decomposition appears repeatedly, the output of one stage is explicitly used by the next stage as a plan, constraint, or scaffold, and the stages have non-uniform criticality. Mathematics and translation satisfy these conditions in our benchmark, but the same principle can be applied to other tasks with similarly stable structure. When any of these properties is weak or unclear, we use the default prefix-first template instead.

C.2 Mathematical Reasoning Template: Plan-and-Solve (PS)

For mathematical reasoning tasks, we decompose the process into a strategic planning stage and a step-by-step execution stage. This aligns with the Plan-and-Solve (PS) prompting method.

Stage 1: Strategy Planning (Executed by M_L)

Instruction: [Input Query]
Let's first understand the problem and devise a plan to solve the problem. Please list the logical steps required, but do not perform the final calculations yet.

Stage 2: Step-by-Step Execution (Executed by M_S)

Instruction: [Input Query]
Plan: [Output from M_L]
Based on the problem and the plan above, let's carry out the plan and solve the problem step by step to find the final answer.

C.3 Translation Template: Deep Reasoning Translation (DRT)

For machine translation tasks, we decompose the process into a terminology-and-constraint identification stage and a full-sentence generation stage. This aligns with the Deep Reasoning Translation (DRT) prompting method. The first stage isolates lexical choices that strongly constrain the final translation, while the second stage produces a fluent full translation under these constraints.

Stage 1: Terminology and Constraint Identification (Executed by M_L)

Instruction: [Input Query]
First identify the key terms, named entities, and domain-specific expressions in the source sentence whose translation is critical to adequacy and consistency. Please provide a concise list of these source expressions together with their preferred translations, but do not generate the full translation yet.

Stage 2: Constraint-Guided Full Translation (Executed by M_S)

Instruction: [Input Query]
Terminology / Constraints: [Output from M_L]
Based on the source sentence and the terminology constraints above, generate a complete, fluent, and faithful translation. Ensure that the specified translations are preserved consistently in the final output.

C.4 Default (Prefix-First) Template

For general-purpose tasks where task-specific structures are latent, we follow a prefix-first prior where M_L establishes a high-quality foundational prefix.

Stage 1: Foundational Prefix (Executed by M_L)

Instruction: [Input Query]
Provide a high-quality, logically sound, and clear initial reasoning or starting portion of the response to the query above.

Stage 2: Contextual Continuation (Executed by M_S)

Instruction: [Full Input Query] + [Prefix Output from M_L]
Continue the response logically based on the provided context and initial reasoning until the completion.

D Baseline hyperparameters

We adopt the recommended configurations from the corresponding papers and official implementations for all baselines. For baselines with calibratable/tunable threshold parameters (e.g., RouteLLM's cost threshold and FrugalGPT's stopping thresholds), we determine these thresholds following the authors' prescribed calibration procedure on a held-out validation set (or a calibration split matched to the test distribution) under the same budget constraint. In particular, RouteLLM's README explicitly recommends threshold calibration and provides example thresholds and commands for reproducing the calibration workflow.

- **KNN-Router:** cosine kNN on prompt embeddings; number of neighbors $k = 40$; embedding model `all-MiniLM-L12-v2`.
- **MLP-Router:** MLP on prompt embeddings; two hidden layers with 100 units each, ReLU; learning rate 10^{-3}; embeddings `all-MiniLM-L12-v2`.
- **RouteLLM:** matrix-factorization router (`mf`); routing controlled by a cost threshold τ calibrated on the validation/calibration split to match the target strong-model usage (budget).

Table 3: GSM8K comparison between the default prefix-first template and the PS template. RouterDC and FrugalGPT are included as the two strongest reference baselines on GSM8K. The best is in bold and the second-best is underlined.

Method	Acc.	Tok.	ρ_L
RouterDC	85.2	881.5	46.4
FrugalGPT	82.7	801.6	44.9
Ours (Default prefix-first)	85.7	519.3	25.0
Ours (PS template)	**87.8**	**407.4**	**23.7**

Table 4: WMT comparison between the default prefix-first template and the DRT template. RouterDC and FrugalGPT are included as the two strongest reference baselines on WMT. The best is in bold and the second-best is underlined.

Method	BLEU	Tok.	ρ_L
RouterDC	37.2	980.4	38.8
FrugalGPT	34.6	690.8	27.6
Ours (Default prefix-first)	37.9	607.4	25.0
Ours (DRT template)	**38.8**	**525.1**	**18.4**

- **RouterDC:** encoder `mDeBERTaV3-base`; AdamW with lr 5×10^{-5} and weight decay 10^{-2}; batch size 64; 1000 training steps; $K^+ = 3$, $K^- = 3$, $H = 3$, $\lambda = 1$, $N = 5$.
- **FrugalGPT:** cascade with a scorer/judger; key parameters are cascade order and stopping thresholds $\{\tau_\ell\}$ calibrated on the validation/calibration split under the same budget constraint.

E Default Prefix-First vs. Specialized Templates on Structured Tasks

E.1 Default Prefix-First vs. PS on GSM8K

To further assess the default template on a task that already admits a domain-specific template, we compare the default prefix-first template with the PS template on GSM8K. We use the same model pair, decoding setup, and evaluation protocol as in the main experiments.

Table 3 shows that the default prefix-first template is already a strong conservative fallback on GSM8K. It outperforms both RouterDC and FrugalGPT, while remaining below the PS template. This is consistent with our template-selection rule: when a task exhibits a stable *plan* $\rightarrow$ *solve* structure, the domain-specific PS template is preferable; when such structure is weak or unclear, the default prefix-first template provides a strong fallback.

E.2 Default Prefix-First vs. DRT on WMT

To further assess the default template on a structured translation task, we compare the default prefix-first template with the DRT template on WMT. We use the same model pair, decoding setup, and evaluation protocol as in the main experiments.

Table 4 shows a similar pattern on WMT. The default prefix-first template already serves as a strong conservative fallback: it outperforms both RouterDC and FrugalGPT, while remaining below the DRT template. This again matches our template-selection rule. When a task exhibits a stable reusable structure—here, terminology grounding followed by constrained translation—the specialized template is preferable; when such structure is weak or unclear, the default prefix-first template remains a competitive task-agnostic fallback.

Taken together, these results clarify the role of the default prefix-first template. It is not intended to replace well-matched domain-specific templates on structured tasks. Rather, it provides a strong conservative fallback that remains competitive even when such structure exists, while specialized templates yield further gains when the task admits a stable reusable stage decomposition.

F Size-Weighted Cost Proxy

Table 1 reports raw token usage (*Tok.*) for transparency, but raw tokens count small-model and large-model tokens equally. To provide a more cost-aware view, we additionally report a size-weighted cost proxy that can be derived directly from the summary statistics in Table 1.

Our true cost definition in Eq. 1 uses per-model input and output token counts with model-dependent coefficients. However, Table 1 reports only the total token usage and the large-model output share ρ_L, rather than the full decomposition into T_L^{in}, T_L^{out}, T_S^{in}, and T_S^{out}. We therefore use ρ_L as a proxy for the large-model token share and define a simple size-weighted proxy with

$$c_S = 1, \qquad c_L = \frac{32}{3} \approx 10.67,$$

Table 5: Size-weighted cost proxy derived from Table 1 using $c_S = 1$ and $c_L = 32/3$. Lower is better. Units can be interpreted as *3B-token equivalents*.

Method	GSM8K	MATH	WMT	AVG
Qwen2.5-3B	211.6	301.2	410.3	307.7
Qwen2.5-32B	10552.5	17577.6	14722.1	14283.7
KNN-Router	1160.0	2812.9	1781.4	1888.0
MLP-Router	1500.1	3038.7	1826.2	2102.5
RouteLLM	3541.4	4427.9	3020.4	3656.5
RouterDC	4835.3	5657.6	4657.6	5060.5
FrugalGPT	4280.8	5546.5	2533.9	4029.5
Ours	**1340.8**	**2313.1**	**1459.1**	**1703.8**

following the model-size ratio between Qwen2.5-3B and Qwen2.5-32B. For routed methods, we compute

$$\widetilde{C} = \text{Tok}\left(1 + \left(\frac{32}{3} - 1\right)\frac{\rho_L}{100}\right).$$

For the two standalone models, we use

$$\widetilde{C}_{3B} = \text{Tok}, \qquad \widetilde{C}_{32B} = \frac{32}{3}\,\text{Tok}.$$

This proxy is not an exact reconstruction of Eq. 1, but it provides a more faithful relative cost view than raw token counts alone.

Under this proxy, our method remains the lowest-cost routing method on all three tasks and in the average view. This is consistent with Table 1: among routing methods, ours combines both low raw token usage and a low large-model output share.