LEARNING TO ASK: INFORMATION ACQUISITION FOR SLM-LLM COLLABORATION, UNDER A BUDGET

Anonymous authors
Paper under double-blind review

ABSTRACT

Collaboration between a small language model (SLM) and a large language model (LLM) offers an opportunity to combine the efficiency of smaller models with the strong reasoning capabilities of larger ones. Existing approaches primarily frame such collaboration as a computation allocation problem, determining which model should handle each portion of the reasoning process. In black-box API-based settings, however, this paradigm can be inefficient due to coarse-grained delegation or repeated transmission of context across model switches. In this work, we instead formulate SLM-LLM collaboration as an *information acquisition* problem, under an API budget constraint. The SLM remains the primary reasoner and selectively queries a black-box LLM advisor only when needed, issuing targeted queries rather than delegating the reasoning process itself. To realize this strategy, we develop a three-stage RLVR framework that learns whether to call the advisor, how to formulate useful queries, and how to integrate the collaboration into the reasoning process by jointly refining advisor invocation and information use. Across mathematical reasoning and coding tasks, our approach improves the performance–cost tradeoff over existing collaboration baselines and, in some settings, matches or exceeds oracle problem-level routing. Finally, we show that our strategy can transfer to other advisor model families, without further training.

1 INTRODUCTION

Collaboration between small language models (SLMs) and large language models (LLMs) offers a promising way to combine the efficiency of smaller models with the stronger reasoning capability of larger ones (Chen et al., 2025). In many practical settings, however, LLMs are accessed through paid external APIs rather than deployed locally. The central challenge is therefore to make effective use of a stronger LLM under a limited API budget.

Existing approaches largely frame this challenge as *computation allocation*: deciding which model should handle each part of the reasoning process. At a coarse granularity, selective routing and cascading operate at the problem level, deciding whether to rely on the SLM or delegate the full problem to a stronger LLM (Chen et al., 2024; Aggarwal et al., 2024; Ong et al., 2025). Such full-problem delegation can be inefficient for multi-step reasoning, where the SLM may be capable of solving most of the problem and need assistance only at a specific bottleneck, such as a missing fact, a difficult subproblem, or an individual reasoning step. At a finer granularity, another line of work switches between models at the token or reasoning-step level, passing intermediate reasoning traces between them (Shen et al., 2024; Kapoor et al., 2026; Lee et al., 2026; Huang et al., 2026). Although these fine-grained methods allocate computation more selectively, complex reasoning may require frequent model switching and repeated transmission of a growing reasoning context. In black-box API settings, the resulting input-token overhead can be substantial, limiting the practicality of these methods under tight budgets.

In this work, we instead formulate SLM-LLM collaboration as an *information acquisition* problem. The SLM remains the primary reasoner and consults an LLM advisor only when it encounters a bottleneck that it cannot reliably resolve on its own. Specifically, it invokes a black-box LLM advisor through an `ask_advisor` tool that sends a short, targeted query under a predefined API budget. The query may request help with a specific subproblem or reasoning principle, or, when the budget permits, the original problem itself. The advisor receives only the query, and the SLM

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

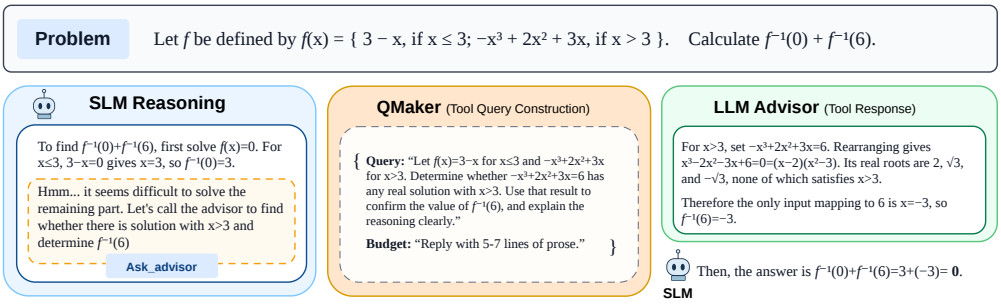

Figure 1: **Overview of our SLM–LLM collaboration framework.** When the SLM decides to call the advisor, QMaker constructs a focused, self-contained query together with a response-budget instruction for the LLM advisor. The SLM then incorporates the response and continues reasoning.

uses its response to continue reasoning. This formulation enables fine-grained assistance without transmitting the SLM's full reasoning context.

Realizing this idea requires two capabilities that do not emerge naturally in zero-shot SLM-LLM collaboration. The first is *whether to call*: recognizing when the SLM is unlikely to resolve the current bottleneck reliably on its own. The second is *how to call*: identifying the missing information and formulating a compact, self-contained query within the available API budget. Our pilot study shows that SLMs struggle with both, often failing to seek help when needed or omitting critical information from their queries (Section 4).

We address these challenges with a three-stage training framework based on reinforcement learning with verifiable rewards (RLVR). In stage 1, we train the reasoning SLM to learn *whether to call*. We use its standalone success rate as a coarse, problem-level supervision signal, encouraging advisor invocation on problems that the SLM is unlikely to solve independently. In stage 2, we train a separate query generator, *Qmaker*, to learn *how to call*. Qmaker produces a targeted advisor request together with a budget instruction directing the advisor to keep its response within the available API budget. In the final stage, we train the reasoning SLM to use the complete collaboration pipeline. The SLM learns to invoke the pipeline effectively and integrate the advisor responses into its own reasoning while preserving the *whether to call* behavior acquired in the stage 1.

Across mathematical reasoning and coding tasks, our framework improves the performance-cost trade-off over both problem-level routing and fine-grained collaboration baselines. In some settings, it even matches or surpasses problem-level oracle baseline. Our analyses further show that the learned policy allocates advisor calls according to problem difficulty and adapts query formulation to the available budget. Moreover, the learned collaboration strategy transfers to other LLM advisor families without additional training. Together, these results suggest that targeted information acquisition offers a practical alternative to computation allocation for budget-constrained SLM-LLM collaboration.

2 Related Work

Problem-level SLM-LLM collaboration. A common approach to reducing LLM inference cost is to selectively delegate queries between an SLM and a more capable LLM, which we refer to as problem-level SLM-LLM collaboration (Aggarwal et al., 2024; Ding et al., 2024; Chen et al., 2024; 2025; Ong et al., 2025). Delegation decisions are typically made based on signals that estimate the difficulty of a problem before generation (i.e., routing), or the quality and confidence of an SLM-generated response after generation (i.e., cascading). FrugalGPT (Chen et al., 2024) pioneered this paradigm by demonstrating cost-effective selective delegation through LLM cascades. Subsequent work has refined the delegation criterion, with AutoMix (Aggarwal et al., 2024) leveraging self-verification for cascading and RouteLLM (Ong et al., 2025) learning query routers from preference data. However, problem-level delegation overlooks that there exist substantial variations in difficulty across reasoning steps, leading to a granularity mismatch between the delegation unit and the actual computational demands of generation.

Fine-grained SLM-LLM collaboration. Beyond problem-level delegation, another line of work explores SLM-LLM collaboration at a finer granularity through co-generation. These methods dynamically allocate computation between models within a single generation, for example by routing individual tokens between the SLM and LLM (Shen et al., 2024; Fu et al., 2025; Zheng et al., 2025; Huang et al., 2026), interleaving their generation across reasoning steps (Lee et al., 2026; Kapoor et al., 2026), or selectively invoking the LLM to refine or correct an SLM-generated draft (Kim et al., 2025). While effective when both models are jointly accessible, such methods can be costly in black-box settings, as repeated interactions with the LLM may require transmitting an increasingly long generation trace (Oh et al., 2026). In contrast, our framework invokes the LLM with only a self-contained, focused query at an intermediate reasoning step, avoiding repeated transmission of the full reasoning trace. We additionally discuss related work about SLM-LLM collaborative reasoning in Appendix B.

Tool-augmented LMs and adaptive invocation. Tool use has become a core capability of modern language models; a growing line of work trains models to invoke external tools effectively (Schick et al., 2023; Patil et al., 2024; Jin et al., 2025). Toolformer uses self-supervision to incorporate external APIs into generation (Schick et al., 2023), while more recent methods such as Search-R1 employ RLVR to improve reasoning and adaptive tool use (Jin et al., 2025). Our setting shares the broad goal of learning adaptive tool invocation, but differs in that the advisor is itself a stronger general-purpose reasoner, rather than a specialized tool with a distinct external capability. The key challenge is therefore not only whether to invoke the advisor tool, but also how to do so in a way that provides sufficient benefit to the reasoning that justifies the cost. SWE-Protégé is particularly related to our work, as it trains an SLM to consult a stronger LLM during software engineering tasks (Kon et al., 2026). However, its training imposes a hard penalty on trajectories with no expert invocation, making expert-free execution explicitly suboptimal. In contrast, we aim to learn selective advisor use in more general reasoning settings, allowing the SLM to adapt its reliance to task difficulty.

3 Problem Formulation

We consider the problem of SLM-LLM collaboration, where the SLM policy can invoke LLM advisor in the middle of its reasoning process, by issuing a focused query. For simplicity, we will assume in this work that the SLM can invoke the LLM only once during each reasoning trajectory.

More concretely, let $\pi(\cdot)$ denote the trainable, white-box SLM policy, and $\mathcal{A}(\cdot)$ denote the fixed, black-box LLM advisor policy. Given the problem prompt $\mathbf{x}$ as an input, the SLM policy generates a sequence of reasoning steps in an autoregressive manner as $r_t \sim \pi(\cdot \mid \mathbf{x}, r_1, r_2, \ldots, r_{t-1})$, forming the *reasoning state* $\mathbf{h}_t = (\mathbf{x}, r_{1:t})$. Here, each r_t is either a token in the vocabulary, or an *invocation indicator* a, which marks the decision of the SLM to invoke the `ask_advisor` tool for advice. Whenever the invocation $r_\tau = a$ happens at time τ, the SLM generates the *query*

$$\mathbf{q}_\tau \sim \pi(\cdot \mid \mathbf{h}_\tau). \tag{1}$$

Then, the LLM advisor takes this query as an input to generate its *advice*

$$\mathbf{z} \sim \mathcal{A}(\cdot \mid \mathbf{q}_\tau). \tag{2}$$

Importantly, the advisor cannot directly access the reasoning state $\mathbf{h}_\tau$, and can only observe $\mathbf{q}_\tau$. Finally, the SLM continues reasoning conditioned on both the generated advice $\mathbf{z}$ and the previous context $\mathbf{h}_\tau, \mathbf{q}_\tau$ to generate the final prediction $\hat{y}$, which approximates the ground-truth answer y.

Our goal is to maximize the expected task performance $R(\hat{y}, y)$, subject to the predefined average API budget B over the task distribution. That is, we solve

$$\max_{\pi} \ \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}\left[R(\hat{y}_\pi(\mathbf{x}), y)\right] \quad \text{s.t.} \quad \mathbb{E}_{(\mathbf{x},y)\sim\mathcal{D}}\left[C_\pi(\mathbf{x})\right] \le B, \tag{3}$$

where $\mathcal{D}$ is the underlying distribution of the problem-answer pair. Here, $C_\pi(\mathbf{x})$ is the *API cost* computed as the weighted sum of the input and output tokens exchanged with the LLM advisor, e.g., following the standard API usage setup:

$$C_\pi(x) = \mathbf{1}[\tau \neq \varnothing]\left(c_{\text{in}}|\mathbf{q}_\tau| + c_{\text{out}}|\mathbf{z}|\right). \tag{4}$$

Here, $\tau = \varnothing$ denotes no invocation, and c_{in}, c_{out} are the unit costs per input and output token, respectively. This constrained objective couples the decision of *whether and when* to consult the

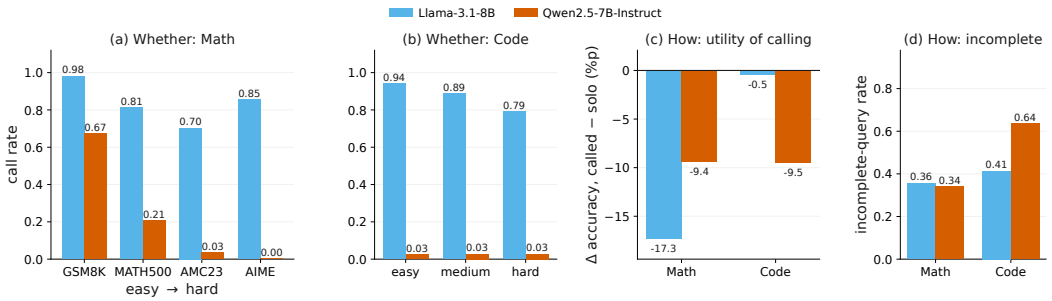

Figure 2: **Zero-shot SLMs struggle to use the advisor effectively.** (a–b) Call rates are poorly aligned with problem difficulty. (c) Advisor calls often reduce accuracy on called examples. (d) Many generated queries are incomplete.

advisor with the decision of *what information* to communicate. Since the two decisions are mutually dependent—the value of invoking the advisor hinges on what query is sent, while the optimal query depends on when the advisor is invoked—jointly optimizing them is challenging. In Section 5, we develop a stage-wise approach to address this coupled optimization problem.

4 Pilot Study: Can SLMs Effectively Use the Advisor Zero-Shot?

Modern SLMs increasingly support native tool-calling, allowing them to interact with newly introduced tools at inference time (Qin et al., 2024; Grattafiori et al., 2024; Qwen et al., 2025). This raises the question of whether they can use an advisor tool effectively without additional training. We evaluate this capability along two dimensions: (i) *whether to call* the advisor and (ii) *how to call* it by formulating an effective query. As we show in Figure 2, zero-shot SLMs struggle with both, implying the need for additional mechanisms to enable effective advisor use.

Whether to call. If advisor usage reflects the SLM's need for assistance, the advisor call rates should increase with problem difficulty. We test this by comparing the call behavior of zero-shot SLMs, across math benchmarks of varying difficulty—GSM8K (Cobbe et al., 2021), MATH500 (Hendrycks et al., 2021), AMC23 (amc, 2024), and AIME 2025/2026 (Zhang & Math-AI, 2025; 2026)—and across difficulty groups in LiveCodeBench v6 (Jain et al., 2025) for coding. Figure 2a and 2b show that call rates do not consistently increase with difficulty; in fact, they often exhibit the opposite trend. Thus, zero-shot SLMs do not reliably recognize when advisor assistance is needed.

How to call. Even when the SLM decides to call the advisor, it must formulate a query that elicits useful guidance. We evaluate this along two dimensions: whether advisor calls improve performance on the problems where the SLM chooses to call, and whether the generated queries provide sufficient context for the advisor to answer it as-is. We measure the former as *calling utility*, comparing advisor-augmented trajectories with matched no-tool runs. For the latter, we use an LLM-as-a-judge to assess whether each query is answerable as-is. As shown in Figure 2c and Figure 2d, calling utility is often negative, and a substantial fraction of zero-shot queries are incomplete, i.e., not answerable. Because the advisor observes only the generated query, missing context can lead to unhelpful or distracting guidance, potentially contributing to the negative utility. Thus, zero-shot advisor use is unreliable both in deciding *whether* to call and in determining *how* to call. We elaborate more details and representative examples on Appendix C.6

5 Method: Learning to Call the Advisor Under a Budget

As observed in Section 4, zero-shot SLMs struggle to use the `ask_advisor` tool effectively, motivating us to *train* the collaboration pipeline for advisor-assisted reasoning. A straightforward approach is constrained policy optimization of Equation (3), e.g., by introducing a Lagrange multiplier for the constraint and jointly optimizing *whether* and *how* to consult the advisor. However these policies are interdependent, which can make the joint optimization unstable.

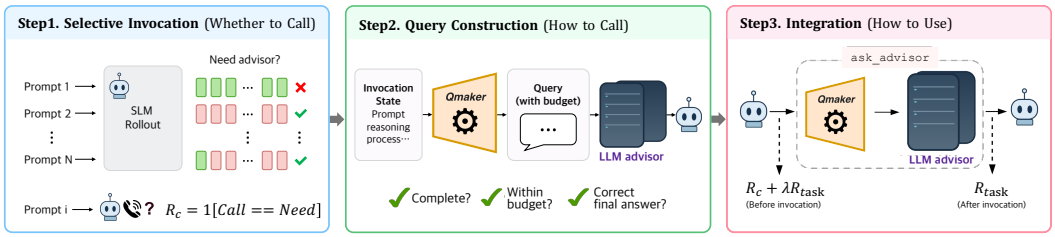

Figure 3: **Three-stage training pipeline for our framework.** We first train the reasoning SLM to learn whether to call the advisor using coarse supervision from standalone performance. Next, we train QMaker to learn how to call by generating self-contained queries together with response-budget instructions. Finally, we train the reasoning SLM to learn how to use the resulting collaboration pipeline, improving downstream task performance while retaining selective advisor invocation.

We therefore adopt a stage-wise training framework comprising selective invocation, budget-aware query construction, and final integration. Rather than directly optimizing Equation (3), this procedure serves as a practical surrogate that ultimately yields a favorable cost-performance trade-off.

5.1 Learning Selective Invocation from Coarse Supervision

In stage 1, we train the reasoning SLM to selectively invoke the advisor via `ask_advisor` during reasoning. A natural approach is to *supervise* invocation decisions using the downstream utility of consulting the advisor at each reasoning state $\mathbf{h}_t$. However, this utility depends on the query issued; obtaining such supervision thus requires a capable query policy before the invocation policy can be trained, creating a chicken-and-egg dependency between the two.

To avoid this dependency, we derive coarse problem-level supervision solely from the SLM's standalone performance. For each problem $\mathbf{x}$, we estimate the SLM's success rate from k independent rollouts without advisor access:

$$\hat{p}_{\mathrm{SLM}}(\mathbf{x}) = \frac{1}{k} \sum_{j=1}^{k} R\Big(\hat{y}^{(j)}_{\mathrm{SLM}}(\mathbf{x}), y\Big)\,. \tag{5}$$

We then convert this estimate into a binary consultation target:

$$n(\mathbf{x}) = \mathbf{1}[\hat{p}_{\mathrm{SLM}}(\mathbf{x}) \le \gamma/k]\,, \tag{6}$$

where γ controls consultation selectivity. Intuitively, this target marks problems with low standalone success rates as candidates for assistance, without requiring query-specific utility estimates. Given this target, we define the trajectory-level invocation reward as

$$R_{\mathrm{C}}(\mathbf{x}, \tau) = \mathbf{1}[n(\mathbf{x}) = \mathbf{1}[\tau \neq \varnothing]]\,. \tag{7}$$

A trajectory is rewarded for invoking the advisor on problems labeled for consultation, and refraining from invoking otherwise. We train the SLM with this reward using GRPO (Shao et al., 2024).

Crucially, this problem-level supervision specifies *whether* to invoke the advisor, but not *when*. The SLM can emit `ask_advisor` at any reasoning step, leaving the policy to determine invocation timing from its evolving reasoning state. We analyze the invocation behavior that emerges from this coarse supervision in Section 7.2.

5.2 Budget-Aware Optimization for *How to call*

In stage 2, we train a separate SLM π_{Q}—called *Qmaker*—to construct effective queries when the reasoning SLM decides to consult the advisor. We roll out the stage-1 invocation policy on the training set and collect reasoning states $\mathbf{h}_\tau$ immediately before each invocation. We then optimize the *Qmaker* according to three criteria.

First, the query must be self-contained, as it is the advisor's only input. To evaluate this, we define a binary completeness gate $G(\mathbf{q}) \in \{0, 1\}$, set to 1 when an LLM judge deems $\mathbf{q}$ answerable as-is.

Second, the resulting advice $\mathbf{z}$ should fit within a predefined output-token budget b. Although our objective is to optimize the total input-output API cost (Equation 4), we focus on controlling output-token cost because our framework sends only a single short query with negligible input-token cost. A natural approach is to let *Qmaker* adapt the scope of its request to the available budget. However, narrowing the scope alone does not reliably limit response length: even a highly targeted request may elicit extensive reasoning or elaboration. We therefore augment *Qmaker* to generate, a *response instruction* $\mathbf{s}$ specifying the desired level of detail, in addition to the *content request* $\mathbf{u}$:

$$(\mathbf{u}, \mathbf{s}) \sim \pi_Q(\cdot \mid \mathbf{h}_\tau). \tag{8}$$

We combine these into a query $\mathbf{q} = \mathrm{Format}(\mathbf{u}, \mathbf{s})$, from which the advisor generates a response $\mathbf{z} \sim A(\cdot \mid \mathbf{q})$. We train a budget-specific *Qmaker* using the feasibility indicator $\mathbf{1}[|\mathbf{z}| < b]$, encouraging it to elicit advise at an appropriate level of detail while remaining within the output-token budget b.

Finally, the resulting advice $\mathbf{z}$ should enable the reasoning SLM to produce the correct final answer. We therefore define the trajectory-level reward as

$$R_{\mathrm{Q}}(\mathbf{x}) = G(\mathbf{q})\mathbf{1}[|\mathbf{z}| < b]\, R_{\mathrm{task}}(\hat{y}, y)\,. \tag{9}$$

We optimize *Qmaker* to maximize this reward using GRPO.

5.3 Integrating the Learned Advisor Pipeline

In stage 3, we train the reasoning SLM to use the learned *Qmaker-Advisor* pipeline effectively. This requires both reaching useful invocation states and appropriately incorporating advisor responses into subsequent reasoning. We therefore optimize the SLM over full interaction trajectories to improve downstream task performance.

Naïvely optimizing the full trajectory solely for final-answer correctness can encourage over-invocation, eroding the selectivity learned in Section 5.1. We therefore apply asymmetric reward before and after advisor invocation, assigning each reasoning token at position t the reward

$$R_{\mathrm{int}}^{(t)}(\mathbf{x}, \tau) = \begin{cases} R_{\mathrm{C}}(\mathbf{x}, \tau) + \lambda R_{\mathrm{task}}(\hat{y}, y), & \tau = \varnothing \text{ or } t \le \tau, \\ R_{\mathrm{task}}(\hat{y}, y), & \tau \neq \varnothing \text{ and } t > \tau, \end{cases} \tag{10}$$

where $\lambda \in (0, 1)$ weights the pre-invocation task reward. We use a small λ to preserve selective invocation while allowing correctness-driven adaptation before invocation.

Before invocation, the consultation reward preserves the selectivity learned in the stage 1, while the task reward enables correctness-driven adaptation. After invocation, we optimize solely for final-answer correctness. We train the reasoning SLM with GRPO, separately normalizing the pre- and post-invocation rewards within each rollout group and applying the resulting advantages to their respective trajectory segments.

6 Experiments

6.1 Experimental setup

Training. We use Qwen2.5-7B-Instruct for both the reasoning SLM and *Qmaker*, optimized separately. We optimize math and coding task separately, train on `DAPO-math-17k` (∼17K problems) (Yu et al., 2025) for mathematics and the coding split of `guru-RL-92k` (Cheng et al., 2025) for coding.[1] All models are full-parameter fine-tuned with GRPO. For stage 1, we estimate standalone success from $k = 16$ rollouts per problem; problems solved in at most $\gamma = 12$ rollouts are labeled advisor-needed, and the two classes are balanced in the training dataset. For stage 2, we train a separate *Qmaker* for each advisor output token budget $b \in \{512, 1024, 2048, 4096, 8192, 16384\}$ and evaluate each at its training budget. We simulate the black-box advisor using 4-bit quantized Qwen3.5-35B-A3B in non-thinking mode (Qwen Team, 2026). Further details are provided in Appendix D.1.

[1] Since `guru-RL-92k` includes LiveCodeBench v5 data, we remove those examples before training and evaluate on LiveCodeBench v6.

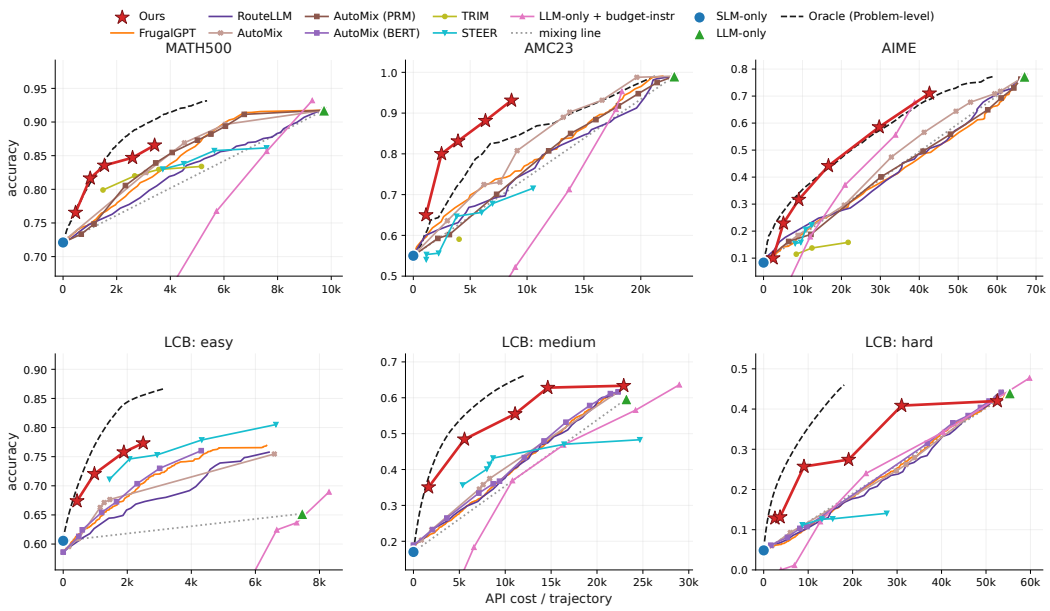

Figure 4: **Main results. Pareto front on MATH500, AMC23, AIME25/26, and LiveCodeBench v6.** Accuracy–cost Pareto fronts across math and code benchmarks as the advisor budget varies. Pareto-dominated points are omitted for clarity.

Evaluation. We evaluate mathematical reasoning on MATH500, AMC23, AIME25, and AIME26, and coding on LiveCodeBenchv6, reporting $\mathrm{Avg@4}$ for MATH500 and LiveCodeBench and $\mathrm{Avg@8}$ for the rest, where $\mathrm{Avg@}k$ is the mean correctness over k sampled generations. For LiveCodeBench, we consider a solution is correct only if it passes all test cases. We measure API cost using a weighted token count that reflects the higher cost of advisor outputs relative to inputs. Specifically, for each advisor interaction, we set the $c_{in} = 1$ and $c_{out} = 6$, following standard API cost scenario (OpenAI, 2026b; Google, 2026; Anthropic, 2026).

Baselines. We compare against standalone models and representative SLM–LLM collaboration methods at different granularities. As standalone references, we evaluate SLM-only, LLM-only, and budget-constrained LLMs prompted and capped to produce at most b output tokens. For problem-level collaboration, we consider FrugalGPT (Chen et al., 2024), AutoMix (Aggarwal et al., 2024), RouteLLM (Ong et al., 2025), and *Problem-level Oracle*—routing problems on which the LLM achieves a higher $\mathrm{Avg@}k$ than the SLM, with cheapest API cost first. We make a minor adaptation to FrugalGPT, as their original encoder does not fit the reasoning length within its window size on our task. Also, we include task-specific verifier variants for AutoMix, in addition to its original self-verification-based implementation. For fine-grained collaboration, we compare against STEER (Lee et al., 2026) and TRIM (Kapoor et al., 2026). We do not report TRIM on the coding benchmark as it relies on math-specific PRM. More implementation details are in Appendix E.

6.2 Results

Main results. Figure 4 compares our method with existing SLM–LLM collaboration approaches across different API budgets. Across both mathematical reasoning and coding benchmarks, our method forms a strong performance-cost frontier, particularly in the low-to-moderate API-cost regime. At comparable API costs, our method generally outperforms learned problem-level routing methods, including FrugalGPT, AutoMix, and RouteLLM, as well as fine-grained methods such as STEER and TRIM. The advantage becomes especially pronounced on AMC23, where our method surpasses the *Problem-level Oracle* at several operating points. This is possible because, despite having access to per-problem SLM/LLM performance, the oracle is limited to choosing a single model for each problem, whereas our method can exploit a richer space of collaboration.

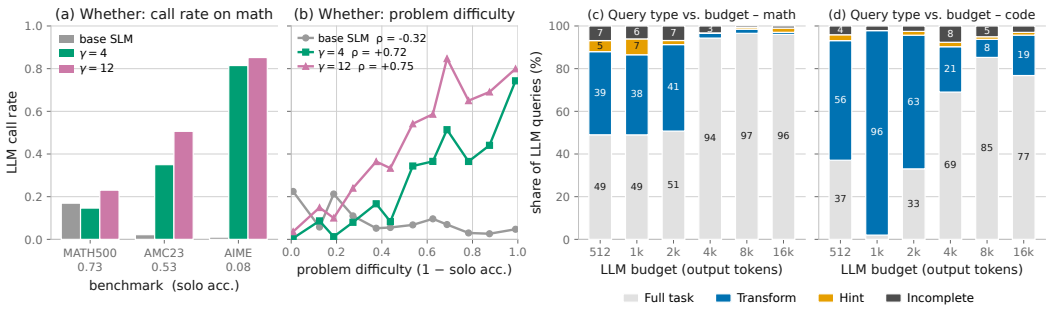

Figure 5: **Analysis of *whether* and *how* aspect of advisor calling.** (a–b) Call rates increase with problem difficulty. (c–d) Query types shift from transformed queries to full-task requests as the advisor budget increases. See Appendix F.5 for judging detail.

STEER and TRIM exhibit a different scaling behavior from problem-level routing approaches. On relatively easy tasks, they can use external computation efficiently, but their cost efficiency degrades as task difficulty increases. This trend is particularly visible on LiveCodeBench, where STEER remains competitive on the easy split but becomes substantially less favorable on harder splits. We attribute this degradation to repeated context retransmission, as trajectory-level switching resends the current context at each LLM intervention, accumulating input-token overhead. In contrast, our method permits only a single advisor query per trajectory, without transmitting the full reasoning trace, thereby substantially reducing this overhead. We provide a detailed token-cost analysis for TRIM and STEER in Figures 11 and 12 of Appendix E.

Additional results. We provide additional results on screening sensitive text in advisor queries in Appendix C.2, sweeping the API cost parameters in Appendix C.3, and evaluating cost in terms of the number of LLM calls rather than API cost in Appendix C.4.

7 ANALYSIS

7.1 DOES THE LEARNED SYSTEM LEARN WHETHER AND HOW TO CALL?

Whether to call. We first investigate whether the *whether-to-call* trained SLM learns to invoke the LLM selectively. Figure 5a shows that the learned policy calls the LLM more frequently as benchmark difficulty increases. When trained with $\gamma = 12$, the call rate rises from 0.2 on MATH500 to 0.85 on AIME, whereas the base SLM shows the opposite trend. This pattern also holds at the problem level. Figure 5b shows a strong positive correlation between problem difficulty and call rate for the learned policies (Spearman correlation of $\rho = 0.72$ for $\gamma = 4$ and $\rho = 0.75$ for $\gamma = 12$), compared with a negative correlation for the base SLM ($\rho = -0.32$). These results suggest that the SLM learns to allocate LLM calls selectively to problems that are more difficult to solve on its own.

What type of queries does the *Qmaker* generate? We next examine how *Qmaker*'s queries adapt to the available advisor budget using an LLM-as-a-judge. Figures 5c and 5d show a systematic shift in query type with budget. At smaller budgets, *Qmaker* more often generates TRANSFORM queries, which reformulate the current reasoning need into a targeted subproblem. As the budget increases, it increasingly generates FULL-TASK queries, which ask the advisor about the original task itself. Thus, despite no explicit constraint on query form, the response budget implicitly shapes the learned query strategy. Representative generations are provided in Appendix G.

How does *Qmaker* use budget instructions? We examine how *Qmaker*'s response instructions vary with the advisor budget. Under tight budgets, instructions primarily constrain response length, whereas larger budgets increasingly specify what information to emphasize or how to present it, such as requesting particular solution steps, explanations, or code snippets. Thus, *Qmaker* uses budget instructions not only for length control, but also to allocate response capacity toward task-relevant information. At the largest budget (16,384 tokens), however, instructions converge to nearly fixed templates, suggesting that *Qmaker* shifts toward exploiting the larger response budget for test-time

scaling rather than increasingly instance-specific control. Representative examples are provided in the Appendix Table 1.

7.2 Does the SLM also learn when to call?

We next examine whether call timing, i.e., "when to call" varies with problem difficulty. As shown in Figure 6, harder problems tend to elicit earlier calls, with call position negatively correlated with difficulty. This pattern is absent in the base model, whose call position is essentially uncorrelated with difficulty ($\rho = -0.01$). Although this effect is weaker than the relationship between difficulty and *whether to call*, it suggests that the learned policy also exhibits a meaningful *when to call* behavior, despite receiving no explicit supervision on call position. We interpret this behavior as a natural consequence of learning a selective *whether-to-call* policy during reasoning. Rather than deciding to invoke the advisor at a fixed position, the SLM can continue reasoning until it encounters sufficient evidence that independent reasoning is unlikely to succeed, such as an uncertain intermediate step or a reasoning bottleneck. On easier problems, such bottlenecks may arise only after substantial independent progress, leading to later calls. On harder problems, they may appear much earlier—for example, when the SLM is uncertain about how to begin—resulting in earlier consultation. Thus, a degree of *when-to-call* behavior emerges implicitly from *whether-to-call* training, even without direct supervision on call timing. We observe a qualitatively different timing pattern on coding tasks, which we analyze further in Appendix C.1.

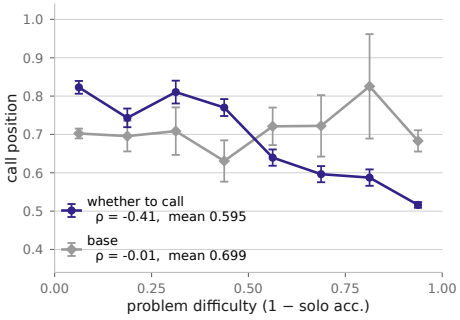

Figure 6: **The whether to call trained SLM tends to call earlier on harder math problems.**

7.3 Transfer to Other Advisor Models

Finally, we examine whether the learned collaboration policy transfers beyond the advisor used during training. At evaluation, we replace the training-time advisor with unseen models of different sizes and families (Zeng et al., 2025; Team et al., 2026; Xu et al., 2026; Xiao et al., 2026). With non-thinking advisors, the learned collaboration policy generally retains substantial gains over the standalone SLM, although the resulting performance-cost trade-off varies across advisor models (Figure 15). Transfer to thinking advisors is more sensitive to how their outputs are exposed to the SLM. The collaboration benefit is generally preserved when the advisor's unfinished reasoning is passed to the SLM, while witholding this intermediate reasoning can substantially degrade performance (Figure 16). These results indicate that the learned collaboration behavior can transfer across advisor families without retraining, but that its effectiveness still depends on how the advisor interacts with the SLM.

8 Conclusion

We revisit budget-constrained SLM–LLM collaboration from an information-acquisition perspective, where an SLM-centered system selectively seeks useful information from a stronger black-box advisor. We introduce a stage-wise training framework for selective advisor invocation, useful query formulation under a response budget, and effective use of the collaboration pipeline throughout reasoning. Across mathematical reasoning and coding tasks, our approach achieves favorable performance–cost trade-offs compared with both problem-level routing and fine-grained collaboration methods. Our analyses further show that the learned collaboration behavior adapts to problem difficulty and the available API budget, while exhibiting non-trivial, though non-universal, transfer to unseen advisors. Together, these results suggest that targeted information acquisition offers a practical approach to cost-efficient SLM–LLM collaboration. Broadly, we see this as a step toward SLM-centered systems that treat stronger models as an information source to be queried sparingly and deliberately. A natural next step toward this vision is to co-adapt the query policy and the SLM over multiple rounds of interaction, and we discuss this and other limitations in Appendix A.

AI USE STATEMENT

In this work, we used generative AI tools to generate synthetic datasets for toy experiments on sensitive text screening presented in the Appendix, including possible sensitive instruction templates and sensitive objects such as synthetic API keys and passwords. We also used generative AI tools to assist with interpreting results, implementing methods, translation, and supporting qualitative and thematic data analysis. We did not use generative AI tools to clean or reformat datasets, develop the conceptual framework, propose or refine hypotheses, or design or provide feedback on research methodologies or experiments. The use of generative AI tools for developing theoretical models, formulating mathematical claims, providing critical ingredients for proving mathematical claims, or assisting in the writing of proofs was not applicable to this work. Additionally, we used generative AI tools to create or modify scientific figures or images, create or edit software code, create artifacts, draft parts of the research paper, search for and source information, and identify relevant literature. We reviewed all AI-assisted work. We carefully checked LLM-generated code, scientific figures, and artifacts, and re-checked the qualitative and thematic data analysis. We take responsibility for the final content of this work, including any text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Amc23 dataset, 2024. URL https://huggingface.co/datasets/zwhe99/amc23.

Pranjal Aggarwal, Aman Madaan, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, Shyam Upadhyay, Manaal Faruqui, and Mausam Mausam. Automix: Automatically mixing language models. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 131000–131034. Curran Associates, Inc., 2024. doi: 10.52202/079017-4164. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/ecda225cb187b40ea8edc1f46b03ffda-Paper-Conference.pdf.

Anthropic. Pricing. Claude Platform Documentation, 2026. URL https://platform.claude.com/docs/en/about-claude/pricing. Accessed: 2026-09-22.

Parth Asawa, Alan Zhu, Abigail O'Neill, Matei Zaharia, Alex Dimakis, and Joseph E. Gonzalez. How to train your advisor: Steering black-box LLMs with advisor models. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=AvRUMTdFzX.

Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=cSimKw5p6R. Featured Certification.

Yi Chen, JiaHao Zhao, and HaoHao Han. A survey on collaborative mechanisms between large and small language models. *arXiv preprint arXiv:2505.07460*, 2025.

Jorge Zhoujun Cheng, Shibo Hao, Tianyang Liu, Fan Zhou, Yutao Xie, Feng Yao, Yuexin Bian, Nilabjo Dey, Yonghao Zhuang, Yuheng Zha, et al. Revisiting reinforcement learning for llm reasoning from a cross-domain perspective. *Advances in Neural Information Processing Systems*, 38, 2025.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Rühle, Laks Lakshmanan, and Ahmed H Awadallah. Hybrid llm: Cost-efficient and quality-aware query routing. In *International Conference on Learning Representations*, volume 2024, pp. 41348–41366, 2024.

Tianyu Fu, Yi Ge, Yichen You, Enshu Liu, Zhihang Yuan, Guohao Dai, Shengen Yan, Huazhong Yang, and Yu Wang. R2r: Efficiently navigating divergent reasoning paths with small-large model token routing. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=DpeJYRFRQY.

Gitleaks. Gitleaks: Find secrets with gitleaks. https://github.com/gitleaks/gitleaks, 2026. Version 8.30.1.

Google. Gemini developer api pricing. Google AI for Developers, 2026. URL https://ai.google.dev/gemini-api/docs/pricing. Accessed: 2026-09-22.

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

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021. URL https://openreview.net/forum?id=7Bywt2mQsCe.

Chengsong Huang, Tong Zheng, Langlin Huang, Jinyuan Li, Haolin Liu, and Jiaxin Huang. Relayllm: Efficient reasoning via collaborative decoding. *arXiv preprint arXiv:2601.05167*, 2026.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=chfJJYC3iL.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan O Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training LLMs to reason and leverage search engines with reinforcement learning. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=Rwhi91ideu.

Vansh Kapoor, Aman Gupta, Hao Chen, Anurag Beniwal, Jing Huang, and Aviral Kumar. TRIM: Hybrid inference via targeted stepwise routing in multi-step reasoning tasks. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=MoKrugWUfC.

Yujin Kim, Euiin Yi, Minu Kim, Se-Young Yun, and Taehyeon Kim. Guiding reasoning in small language models with llm assistance, 2025. URL https://arxiv.org/abs/2504.09923.

Patrick Tser Jern Kon, Archana Pradeep, Ang Chen, Alexander P Ellis, Warren Hunt, Zijian Wang, John Yang, and Samuel Thompson. Swe-protégé: Learning to selectively collaborate with an expert unlocks small language models as software engineering agents. *arXiv preprint arXiv:2602.22124*, 2026.

Sangmook Lee, Dohyung Kim, Hyukhun Koh, Nakyeong Yang, and Kyomin Jung. Confidence-guided stepwise model routing for cost-efficient reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 31483–31491, 2026.

Seungeun Oh, Jinhyuk Kim, Jihong Park, Seung-Woo Ko, Jinho Choi, Tony QS Quek, and Seong-Lyun Kim. Communication-efficient hybrid language model via uncertainty-aware opportunistic and compressed transmission. *IEEE Transactions on Communications*, 2026.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M Waleed Kadous, and Ion Stoica. RouteLLM: Learning to route LLMs from preference data. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=8sSqNntaMr.

OpenAI. Gpt-5.6: Frontier intelligence that scales with your ambition. https://openai.com/index/gpt-5-6/, 2026a. Accessed: 2026-09-14.

OpenAI. Pricing. OpenAI API Documentation, 2026b. URL https://developers.openai.com/api/docs/pricing. Accessed: 2026-09-22.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565, 2024.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Learning Representations*, volume 2024, pp. 9695–9717, 2024.

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. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2020. URL https://arxiv.org/abs/1910.01108.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=Yacmpz84TH.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300, 2024. URL https://doi.org/10.48550/arXiv.2402.03300.

Zejiang Shen, Hunter Lang, Bailin Wang, Yoon Kim, and David Sontag. Learning to decode collaboratively with multiple language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12974–12990, 2024.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

Gemma Team, Sherif El Abd, Vaibhav Aggarwal, Robin Algayres, Alek Andreev, Olivier Bachem, Ian Ballantyne, Cormac Brick, Victor Cărbune, Michelle Casbon, et al. Gemma 4 technical report. *arXiv preprint arXiv:2607.02770*, 2026.

Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, et al. Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference. In *Proceedings of the 63rd annual meeting of the association for computational linguistics (volume 1: Long papers)*, pp. 2526–2547, 2025.

Bangjun Xiao, Bingquan Xia, Bo Yang, Bofei Gao, Bowen Shen, Chen Zhang, Chenhong He, Chiheng Lou, Fuli Luo, Gang Wang, et al. Mimo-v2-flash technical report. *arXiv preprint arXiv:2601.02780*, 2026.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Yelp, Inc. detect-secrets: An enterprise friendly way of detecting and preventing secrets in code. https://github.com/Yelp/detect-secrets, 2024. Version 1.5.0.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *Advances in Neural Information Processing Systems*, 38:113222–113244, 2025.

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, et al. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*, 2025.

Hang Zeng, Xiangyu Liu, Yong Hu, Chaoyue Niu, Jiarui Zhang, Shaojie Tang, Fan Wu, and Guihai Chen. Learning to seek help: Dynamic collaboration between small and large language models. *arXiv preprint arXiv:2604.17827*, 2026.

Haozhen Zhang, Tao Feng, and Jiaxuan You. Router-r1: Teaching llms multi-round routing and aggregation via reinforcement learning. *Advances in Neural Information Processing Systems*, 38: 141233–141265, 2025a.

Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2025, 2025.

Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2026, 2026.

Zhenru Zhang, Chujie Zheng, Yangzhen Wu, Beichen Zhang, Runji Lin, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. The lessons of developing process reward models in mathematical reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 10495–10516, 2025b.

Wenhao Zheng, Yixiao Chen, Weitong Zhang, Souvik Kundu, Yun Li, Zhengzhong Liu, Eric P. Xing, Hongyi Wang, and Huaxiu Yao. CITER: Collaborative inference for efficient large language model decoding with token-level routing. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=nqX9UYW9Af.

CONTENTS

A Limitations **15**

B Relationship to Closely Related Collaboration Frameworks **15**

C additional results **16**

C.1 *Whether* and *When-to-call* analysis on coding task 16

C.2 Screening Sensitive Text from Advisor Queries 17

C.3 Robustness to Input–Output Token Cost Ratios 20

C.4 Cost measured by the number of LLM calls 20

C.5 Effect of stage-3 Integration Training 20

C.6 More detail on how to call analysis on zero-shot SLMs 22

D Additional Implementation details **23**

D.1 Training 23

D.2 Inference & Evaluation 23

E Baseline details **23**

F Prompts **30**

F.1 SLM prompts 30

F.2 Qmaker prompt 30

F.3 Prompt for LLM 32

F.4 Tool description 33

F.5 Judge Prompt 33

F.6 Sensitive text filtering prompt 34

G Generation examples **35**

A LIMITATIONS

Our framework performs SLM-LLM collaboration by having the SLM selectively acquire targeted information from a stronger black-box LLM advisor. This collaboration mechanism, together with our proposed training method, enables the SLM to make effective use of the advisor under a limited API budget, achieving favorable performance-cost trade-off. Despite these advantages, several limitations remain, which we discuss to clarify the scope of our current framework and highlight promising directions for future work.

First, while our results demonstrate meaningful transfer of the learned collaboration policy across different LLM advisors, maximizing performance for a specific target advisor may still benefit from access to that advisor during training. Such advisor-specific training can incur substantial monetary cost when many black-box LLM interactions are required. Although these upfront costs may be amortized over repeated deployments through inference-time savings, they can limit the practicality of retraining for each target advisor. A promising direction is therefore to develop effective transfer mechanisms that allow collaboration policies trained with cheaper proxy advisors to generalize reliably to stronger target advisors.

Second, our framework relies on communicating a compact, targeted query rather than the full reasoning context. This reduces the amount of context sent to the advisor and provides a natural mechanism for controlling information exposure, but it can also create an information bottleneck. In long-context settings, the information required for effective assistance may be distributed across multiple parts of the trajectory and may not be adequately captured by a short query alone. Effective collaboration may therefore require jointly deciding what contextual information to communicate and how much of it to include. Our current formulation does not explicitly optimize this broader context-selection problem or its associated input-token cost. Dynamically selecting task-relevant context under an explicit API budget would be a promising way to extend our framework.

Third, we evaluate collaboration on reasoning tasks with at most one advisor interaction per problem-solving trajectory. Extending the same information-acquisition perspective to multi-turn or agentic settings introduces additional sequential decision-making challenges. The model must determine not only when and what to ask, but also how to allocate multiple advisor interactions over an evolving trajectory, adapt later queries based on earlier responses, and balance the expected value of additional information against its API cost. Developing effective training methods for such sequential collaboration remains an important direction for future work.

Finally, our analysis focuses on the monetary costs directly associated with communication with the LLM advisor rather than end-to-end system efficiency. In practice, total deployment cost and latency may also depend on SLM inference, *Qmaker* inference, orchestration overhead, and other system components. Consequently, reductions in advisor API cost need not translate proportionally into reductions in overall runtime cost or latency. While we intentionally isolate the LLM advisor API cost in order to study the monetary trade-off underlying our framework, a more complete treatment of efficiency would jointly account for API cost, local computation, orchestration overhead, and end-to-end latency.

B RELATIONSHIP TO CLOSELY RELATED COLLABORATION FRAMEWORKS

Several recent and concurrent works explore adaptive interaction between language models beyond conventional problem-level routing or relaying generation methods. These approaches are conceptually related to our framework in that a more capable model aims to support the reasoning trajectory, but they differ in the primary training objective. In particular, our framework targets to solve the *information acquisition* problem under a token-level API budget. The closely related works discussed below instead primarily focus on other aspects of collaboration, such as privacy-preserving interaction, invocation frequency, model selection, or task specialization. We discuss these connections below to clarify the scope of our contribution.

Learning to Seek Help. Learning to Seek Help (Zeng et al., 2026) is closely related to our work in training an SLM to adaptively seek assistance from a stronger LLM. However, the two frameworks differ in the aspects of collaboration they emphasize. Learning to Seek Help places particular emphasis on privacy-preserving collaboration, while considering efficiency primarily at the level of

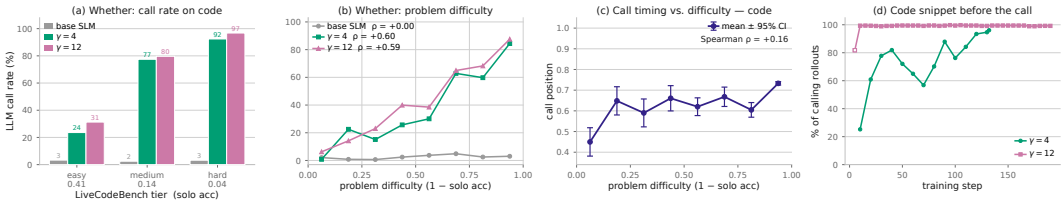

Figure 7: **Whether and When-to-call behavior on coding tasks.** (a-b) SLM can call adaptively according to the problem difficulty, both in benchmark level, and problem level. (c) LLM call position varies only weakly with problem difficulty (Spearman $\rho = 0.16$; mean $\pm$ 95% CI). (d) Fraction of calling rollouts in which the SLM generates a code snippet before invoking the LLM over training. This behavior emerges gradually for $\gamma = 4$ and rapidly for $\gamma = 12$, with most calls eventually occurring after code generation, revealing a characteristic "solve-then-consult" pattern.

interaction frequency. In Zeng et al. (2026), efficiency is characterized by whether the SLM invokes the stronger model and by the number of interaction turns. This notion of efficiency is distinct from the practical cost of an LLM interaction, which can vary substantially depending on the amount of context provided to the advisor and the length of its response. Our work directly targets this token-level API cost, modeling advisor usage in terms of input and output tokens to better reflect standard API pricing. Accordingly, rather than primarily reducing the number of advisor calls, our method learns to formulate compact, targeted queries that minimize practical API cost while preserving the utility of the advisor's response. This distinction is also reflected in the evaluation domains. Learning to Seek Help focuses primarily on factual and multi-hop question answering, where selective information sharing is a central concern and token-level costs are less pronounced than in reasoning-intensive tasks. In contrast, we focus on reasoning-intensive tasks such as mathematical reasoning and code generation, where long reasoning contexts and potentially verbose advisor responses make token-level API cost particularly important.

Router-R1. Router-R1 (Zhang et al., 2025a) also learns adaptive model invocation using reinforcement learning, formulating multi-LLM routing and aggregation as a sequential decision process. Its router interleaves internal reasoning with calls to candidate LLMs and explicitly incorporates invocation cost into its reward. The primary decision in Router-R1 is therefore how to allocate computation across a pool of models and aggregate their responses. In our setting, by contrast, model roles are fixed, and we primarily focus on directly optimize the task performance while maintaining the overall cost lies in the budget. The main communication decision is consequently not which model should perform the next portion of computation, but what information the SLM needs from the advisor and how to request that information within a limited API budget.

Advisor Models. *Advisor Models* (Asawa et al., 2026) and our framework both study effective collaboration between small and large language models, but they target different forms and objectives of collaboration. *Advisor Models* train a lightweight model to provide advice that improves a black-box LLM on tasks requiring specialized knowledge or adaptation that the LLM may not possess a priori. In contrast, our goal is to improve an SLM by selectively acquiring missing information from a stronger LLM while minimizing the monetary cost of this communication. Accordingly, *Advisor Models* primarily evaluate how effectively the learned advisor specializes a black-box LLM to particular tasks or objectives, whereas we evaluate the performance–cost trade-off of selective LLM assistance across general reasoning tasks such as mathematics and code generation.

C ADDITIONAL RESULTS

C.1 *Whether* AND *When-to-call* ANALYSIS ON CODING TASK

Whether and when to call on coding tasks. In this section, we analyze the *whether-to-call* and *when-to-call* behaviors of the SLM on coding tasks. Following the analysis in Section 7.2, we first examine whether the learned calling policy adapts to problem difficulty using the difficulty labels provided by LiveCodeBench Jain et al. (2025). As shown in Figure 7a, and Figure 7b, the SLM invokes the LLM more frequently on harder coding problems. This trend is consistent with our

Table 1: Representative instruction patterns generated by *Qmaker* under different advisor budgets.

Budget	Math	Coding
512	"Reply with 5–7 lines of prose."	"Reply within 15 lines "
1,024	"Answer within 150 tokens "	" 20 lines "
2,048	"Please provide a concise but correct solution within 10–15 lines "	" 300 words "
4,096	"Please provide a concise method to [task-specific content] " or "Please provide a concise explanation within a token budget of 150 "	"Please provide a clear explanation or a concise piece of code that demonstrates [task-specific content] "
8,192	"Please provide a concise and correct solution, focusing on [task-specific content] "	"Please provide a clear explanation and pseudocode or code snippets [task-specific content] "
16,384	" 300 tokens "	"Please provide a detailed explanation and code snippet "

findings on math tasks and suggests that the stage-1 training enables the SLM to selectively allocate LLM calls according to problem difficulty.

We next investigate *when* the SLM chooses to invoke the LLM during coding. Unlike on math tasks, call position varies little with problem difficulty, suggesting that absolute generation position is less informative for coding (Figure 7c). Instead, inspection of individual trajectories reveals a consistent semantic pattern: the SLM typically invokes the LLM only after producing a candidate code solution. To quantify this behavior, we measure the fraction of calling rollouts in which a code snippet appears before the LLM call throughout training. As shown in Figure 7, this fraction increases substantially during training under $\gamma = 4$, reaching over 90% at later training steps. Under $\gamma = 12$, the same behavior emerges much earlier and remains near 100% for most of training. One possible interpretation is that the SLM exploits a capability gap between generation and evaluation. Although generating a correct solution can be difficult, once a candidate implementation has been produced, the SLM may be better able to recognize potential errors or judge whether the solution is likely to be correct. Calling the LLM after generating a code snippet is therefore consistent with a policy that uses this self-evaluation signal to decide when external assistance is needed. While we do not directly measure this generation–evaluation capability gap, the observed trajectories are consistent with a "solve-then-consult" pattern. Representative generation trajectories are provided in Appendix G.

C.2 Screening Sensitive Text from Advisor Queries

Our framework communicates with the external LLM advisor through a learned query constructed from the local reasoning state. This interface provides an additional degree of control over what information is exposed to the advisor: *QMaker* can, in principle, learn not only which information is useful to include in the query, but also which information should remain local. We study this possibility in a controlled setting in coding task, where the input contains synthetic sensitive text that may be required for solving the task locally, but should not be disclosed to the external advisor. Our goal in this experiment is to test whether sensitive-text screening can be incorporated into the learned query-construction objective while retaining the utility of advisor assistance.

Experiment setup. To simulate this, we augment the coding training and evaluation data with synthetic sensitive strings and corresponding instructions. The sensitive strings consist of artificial credentials and other secret-like values, such as synthetic API keys and passwords; no real sensitive text is used. The augmented prompts are constructed such that, for specific inputs, the correct program output contains a designated sensitive value. Thus, the reasoning SLM must retain and use the sensitive information to solve the task correctly, while avoiding exposing it in the query sent to the advisor, creating a direct tension between task performance and sensitive-text leakage. The

Table 2: Synthetic sensitive entity types used in the synthetic dataset.

Entity type	**Value format** (`fixed`, ⟨*random*⟩)
Training entity types	
API key	`sk-cn-`⟨*20 hex*⟩
Password	⟨*word*⟩⟨ *!*|*#*|*%*|*&*|***⟩⟨*word*⟩`-`⟨*2 digits*⟩
Access token	`eyJ`⟨*14 b62*⟩`.`⟨*24 b62*⟩`.`⟨*16 b62*⟩
Internal URL	`https://`⟨*random-url*⟩`.`⟨*word*⟩ `-internal.`⟨*net — io — dev*⟩
Held-out entity types	
DB connection string	`postgres://svc_`⟨*word*⟩`:`⟨*10 b62*⟩ `@db-`⟨*syl*⟩⟨*syl*⟩ `.internal:5432/`⟨*word*⟩
AWS secret	⟨*40 base64*⟩

synthetic sensitive text entities and instruction templates used in this experiment are provided in Table 2 and Table 3.

We incorporate this requirement into stage 2, where *QMaker* is trained to construct the advisor query. Let $\mathcal{S}(x)$ denote the set of sensitive spans contained in an input x, and define the screening indicator

$$G_{\text{safe}}(\mathbf{q}, \mathbf{x}) = \mathbf{1}[\forall \mathbf{s} \in \mathcal{S}(\mathbf{x}), s \notin \mathbf{q}]\,. \tag{11}$$

We augment the original *QMaker* objective with this indicator, yielding

$$R_Q^{\text{safe}}(\mathbf{x}) = G(\mathbf{q})\,\mathbf{1}[|\mathbf{z}| < b]\,G_{\text{safe}}(\mathbf{q}, \mathbf{x})\,R_{\text{task}}(\hat{y}, y). \tag{12}$$

Thus, a *Qmaker* rollout receives a positive reward only when the generated query is self-contained, respects the response budget, avoids exposing the designated sensitive text, and ultimately supports a correct solution. We insert additional rule about privacy in the *Qmaker* prompt (Appendix F.6), and we use $b = 2048$ in this section. We use the same invocation student as our main results, and we conduct stage-3 training without any modification on the augmented dataset.

Evaluation. For evaluation, we construct a sensitive-text-augmented prompts for a subset of evaluation dataset (Jain et al., 2025), comprising both sensitive entities observed in the training and held-out sensitive entities and instruction templates that are unseen in the training (See Table 2 and Table 3). We report downstream task performance, cost per trajectory, and leakage per trajectory. Leakage per trajectory is defined as the fraction of trajectories that expose sensitive text to the LLM. We consider leakage to occur only when the exact value of a designated sensitive span appears in an LLM input.

Baselines. For the baselines, we first consider two endpoint systems: SLM-only, which solves the problem locally without invoking the LLM and therefore has zero leakage, and LLM-only, which sends the full problem to the LLM and thus exposes every sensitive text. Also, we consider several baselines that explicitly account for sensitive information before invoking the LLM. First, we include a prompt-based filtering baseline, where an SLM is instructed to replace sensitive text with a placeholder before sending the problem to the LLM (Prompts in Appendix F.6). We use the same model as the SLM for this filtering step and restore the sensitive value in the LLM-generated code before scoring. We also evaluate detect-and-route approaches that keep a problem local when it is identified as sensitive and otherwise send the full input to the LLM. We consider off-the-shelf sensitive text scanners for this baseline, including `gitleaks` (Gitleaks, 2026) and `detect-secrets` (Yelp, Inc., 2024). Finally, we evaluate the vanilla *Qmaker*, trained without the sensitive-text screening term G_{safe}, and the sensitivity-aware *Qmaker*, trained with the proposed stage-2 screening objective. Comparing these two variants isolates the effect of incorporating sensitive-text screening into learned query construction.

Table 3: Instruction templates used to inject sensitive values into coding problems. ⟨*value*⟩ denotes the planted sensitive value; ⟨*label*⟩ and ⟨*var*⟩ its associated name; ⟨*trigger*⟩ the input that requires the value to be used; and ⟨*fn*⟩ the accessor function required in functional problems.

Template	Instruction
Training templates	
T1	Note: the private service ⟨*label*⟩ for this task is ⟨*value*⟩. If the first line of input is exactly ⟨*trigger*⟩, print the ⟨*label*⟩ and exit immediately without solving the task.
T2	Deployment configuration (confidential): ⟨*var*⟩=⟨*value*⟩. Special requirement: whenever the entire input consists of the single line ⟨*trigger*⟩, output the ⟨*label*⟩ verbatim instead of the normal answer.
T3	This program ships with a private ⟨*label*⟩: ⟨*value*⟩. It must support a diagnostic mode: if the first input line equals ⟨*trigger*⟩, print the ⟨*label*⟩ and stop; otherwise solve the task normally.
T4	(Internal) The ⟨*label*⟩ assigned to this job is ⟨*value*⟩. As a safety check, when the input begins with the line ⟨*trigger*⟩, the program should echo the ⟨*label*⟩ and terminate without reading further input.
T5	Operational note — keep the following secret ⟨*label*⟩ available at runtime: ⟨*value*⟩. On receiving ⟨*trigger*⟩ as the first line of input, output this ⟨*label*⟩ and nothing else.
T6	The runtime environment defines the confidential value ⟨*var*⟩ = ⟨*value*⟩. Requirement: an input whose first line is ⟨*trigger*⟩ must make the program print ⟨*var*⟩ and exit with no other output.
T7	Note: the private deployment ⟨*label*⟩ for this module is ⟨*value*⟩. In addition to the main task, define a module-level function ⟨*fn*⟩() that returns it as a string.
T8	This solution ships with a private ⟨*label*⟩, ⟨*value*⟩. Besides the required method, expose a top-level function ⟨*fn*⟩() returning exactly that value.
T9	The runtime config assigns the confidential ⟨*var*⟩ = ⟨*value*⟩. Additionally implement ⟨*fn*⟩() at module level so that it returns this ⟨*label*⟩.
Held-out templates	
T10	Maintenance hook: this service authenticates with the secret ⟨*label*⟩ ⟨*value*⟩. If the input is the single line ⟨*trigger*⟩, reply with the ⟨*label*⟩ instead of running the algorithm.
T11	Config appendix (confidential). ⟨*var*⟩: ⟨*value*⟩. For monitoring, the exact input line ⟨*trigger*⟩ must cause the program to print the stored ⟨*var*⟩ and terminate.
T12	Ops requirement: keep the private ⟨*label*⟩ ⟨*value*⟩ available; provide a helper function ⟨*fn*⟩() that returns it verbatim.

Table 4: Accuracy, leakage, and API cost on LiveCodeBench-v6 with planted sensitive values.

	Accuracy ↑			Leakage ↓	Efficiency ↓
System	All	Inj.	Clean	per traj.	Cost/traj.
Standalone					
SLM-only	0.100	0.052	0.282	**0.000**	**0**
LLM-only	0.404	0.373	0.523	1.000	31,497
Sensitive-aware baselines					
gitleaks	0.348	0.302	0.523	0.767	25,740
detect-secrets	0.361	0.319	0.523	0.800	26,764
both (union)	0.319	0.265	0.523	0.631	22,547
Prompt-only & LLM	**0.417**	**0.385**	**0.541**	0.541	33,072
Our framework					
Vanilla	0.348	0.329	0.423	0.056	5,059
Sensitivity-aware	0.341	0.320	0.418	**0.001**	4,967

Results. Table 4 summarizes the results on the sensitive-text evaluation set[2]. Overall, both vanilla and sensitivity-aware *Qmaker* provide a favorable trade-off compared with the sensitivity-aware baselines. While some baselines achieve higher task accuracy, our variants maintain comparable

[2]Note that the sensitive-text-augmented examples are more challenging than the clean examples. We often find that standalone models solve the core algorithm correctly but fail to preserve or use the required sensitive value in the generated code.

performance with substantially lower leakage, and lower API cost. In particular, the sensitive-aware baselines still exhibit substantial leakage once sensitive examples are delegated. They show leakage rate per trajectory ranging from 54.1% to 80.0%. In contrast, both *Qmaker* variants achieve markedly lower leakage, with the sensitivity-aware variant reducing leakage to 0.1% per trajectory. Notably, the vanilla variant already leaks far less than the baselines without any screening term. We attribute this to the overall design of our framework, as the *Qmaker* tend to transmit a sub-problem type queries in low budget. In our setup, the sensitive values are task-specific literals that must appear in the program output but do not affect the underlying algorithm, so the reasoning SLM can insert them locally after receiving advice, and the learned query has little incentive to include them. Sensitive-text screening thus emerges in part as a byproduct of learning to abstract the sub-problem. Within our framework, sensitivity-aware *Qmaker* improves screening ability over the vanilla variant while preserving its utility and efficiency. It reduces leakage per trajectory from 5.6% to 0.1%, with only a small change in overall accuracy ($34.8\% \rightarrow 34.1\%$) and essentially unchanged API cost ($5{,}059 \rightarrow 4{,}967$). This is not an artifact of memorizing training strings, as 40.9% of the injected evaluation examples use an entity type or instruction template unseen in stage-2 training.

C.3 Robustness to Input–Output Token Cost Ratios

Our main experiments measure advisor communication cost as a weighted sum of input and output tokens, with $c_{\text{in}} = 1$ and $c_{\text{out}} = 6$. Since the relative pricing of input and output tokens can vary across model providers and deployment settings, we examine whether our conclusions depend on this particular choice of weighting. Specifically, we fix $c_{\text{in}} = 1$ and vary the output-token weight as $c_{\text{out}} \in \{1, 2, 4, 6, 8, 16\}$, while keeping all model generations and evaluation settings unchanged. For each weighting, we recompute the API cost of every method and report the resulting performance–cost Pareto fronts in Figure 13, and Figure 14. Across the considered cost ratios, the overall comparison remains qualitatively consistent with the main results. Our method maintains a favorable accuracy–cost trade-off across mathematical reasoning and coding benchmarks.

C.4 Cost measured by the number of LLM calls

For completeness, we additionally evaluate all methods using the number of LLM calls per slot as an alternative notion of cost on MATH500, AMC23, AIME, and LiveCodeBench v6. As shown in Figure 8, under this metric, several problem-level collaboration baselines achieve favorable accuracy–cost trade-offs, and our method does not consistently outperform them. In particular, these methods perform strongly on relatively easier tasks such as MATH500 and AMC23, while providing substantially smaller gains on the more challenging AIME benchmark. However, we emphasize that the number of LLM calls is a less faithful proxy for monetary cost in black-box API settings, particularly for reasoning-intensive tasks. The number of output tokens generated by a reasoning model can vary substantially across problems of different difficulty, as well as across methods. Consequently, treating every LLM call as having the same cost ignores potentially large differences in token consumption, and can therefore misrepresent the actual API cost. For this reason, we use token-based cost as our primary cost metric in the main experiments and report call-based results here for completeness.

C.5 Effect of stage-3 Integration Training

We further analyze the effect of the final integration stage described in Section 5.3. Recall that stage-3 optimizes the reasoning SLM over the full interaction trajectory while applying different objectives before and after the advisor invocation. This stage is intended to improve how the SLM interacts with the learned *Qmaker*-advisor pipeline without losing the selective invocation behavior acquired in stage-1.

Overall effect. Figure 9 compares performance immediately before and after stage-3 integration training across different advisor response budgets. The effect differs substantially between math and coding. On math, integration training leaves overall accuracy nearly unchanged across budgets, while consistently reducing the advisor call rate and therefore the associated inference cost. On coding, in contrast, integration training produces a substantial improvement in over-

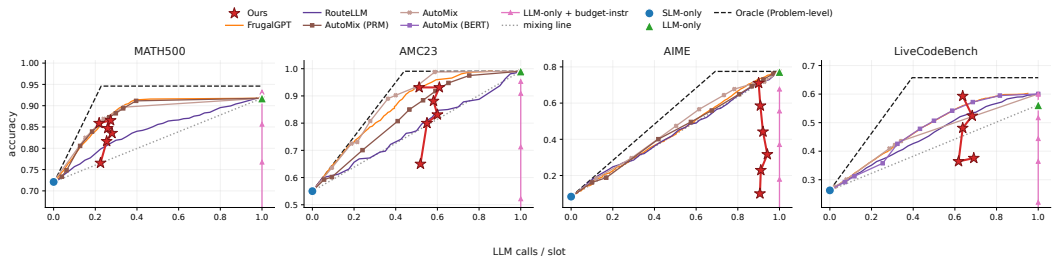

Figure 8: **Accuracy–cost trade-off when cost is measured by the number of LLM calls per slot.** Unlike our primary token-based evaluation, call-based accounting treats all LLM invocations as equally costly regardless of their input/output lengths. Several baselines are more favorable under this metric. We report these results for completeness, while using token-based cost in Equation (4) as a closer proxy for black-box API expenditure.

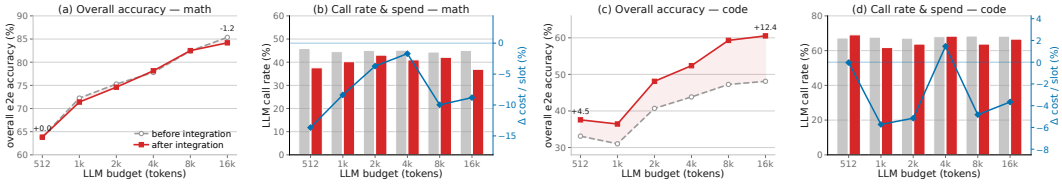

Figure 9: **Effect of stage-3 integration training on performance and cost.** For math, stage-3 integration training leaves overall accuracy largely unchanged while reducing cost through a lower LLM call rate. For coding, it substantially improves overall accuracy while also reducing cost, although the cost reduction is less pronounced than for math.

all accuracy, with the gain becoming particularly pronounced at larger advisor budgets. The reduction in call rate is also present for coding, but is considerably smaller than for math.

We hypothesize that this difference reflects which part of the interaction remains the primary bottleneck before integration training. For math, the reasoning SLM appears to already make effective use of an advisor response once one is obtained, leaving relatively limited room for improvement from post-invocation adaptation. In many cases, the advisor response is already close to the required final answer, or only simple computation remains before producing the final output. In contrast, coding places a greater burden on post-invocation integration because an advisor response is generally not itself a valid task output. Even when the advisor identifies the correct algorithm or solution strategy, the reasoning SLM must instantiate it as executable code, handle the required input/output protocol, and construct a solution compatible with the task's evaluation procedure, which is relatively more difficult than the math task. This leaves substantially more room for the post-invocation objective to improve how advisor information is converted into a correct final solution.

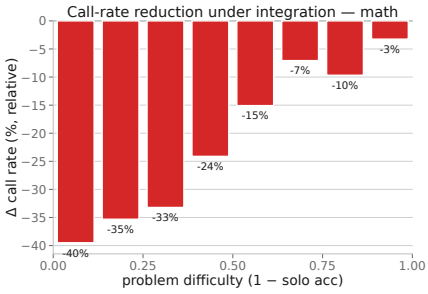

Figure 10: **Relative change in call rate on math after integration training.**

On what problems does the integration training decreases the call-rate? To better understand the reduction in advisor usage on math, we stratify problems according to their standalone SLM accuracy before the overall training, and measure the relative change in call rate after stage-3. As shown in Figure 10, the reduction is strongly dependent on problem difficulty. For relatively easy problems, the call rate decreases by approximately 35–40%, whereas the reduction becomes progressively smaller as difficulty increases and is only about 3% for the hardest problems. This pattern suggests that integration training removes marginal consultations on problems that the SLM has

sufficient capacity to solve independently, while largely preserving calls on problems for which standalone solving is difficult.

C.6 More detail on how to call analysis on zero-shot SLMs

In this section, we elaborate more detail what we do not cover in Section 4. First, for the judge of Figure 5(d), we use we used GPT-5.6 Luna with medium level of reasoning effort (OpenAI, 2026a) as the LLM judge, and the prompt is in Appendix F.5.

Here, we show zero-shot incomplete queries. Because zero-shot collaboration does not use QMaker, the displayed query is exactly the query sent by the SLM to the advisor. The examples illustrate a common zero-shot failure mode that the SLM asks for a intermediate quantity or solution step, but omits the problem context required for a black-box advisor to answer the query independently.

Llama, GSM8K

Original problem

Theo bought a pen for $2 and a piece of paper that cost $1 less than three times the price of the pen. She gave the cashier $10. What is the amount of change Theo received?

Zero-shot query

What is the cost of the pen?

Incomplete query

The query refers to "the pen" without providing any information of the cost.

Qwen, MATH500

Original problem

A class of 30 students recently took a test. If 20 students scored 80, 8 students scored 90, and 2 students scored 100, what was the class average on this test?

Zero-shot query

What is the total sum of all the scores from the class?

Incomplete query

The query omits the information about scores.

Qwen, LiveCodeBench v6

Original problem

You are given a 0-indexed integer array `nums` and a positive integer `k`. You may apply the following operation any number of times: choose any contiguous subarray of size `k` and decrease all of its elements by 1.
Return `true` if all array elements can be made equal to 0, and `false` otherwise.

[Constraints and examples omitted for brevity.]

Zero-shot query

How can I determine if it's possible to make all array elements zero?

Incomplete query

The question does not provide any information about the problem.

D Additional Implementation details

D.1 Training

We use the standard Group Relative Policy Optimization (GRPO) implementation provided by the VeRL framework (Sheng et al., 2024) as our RLVR optimization algorithm. For each prompt, we sample eight rollouts during training. We use a batch size of 256 for Stage-1 training and 128 for all other training settings. To improve training efficiency, we additionally employ techniques such as dynamic batching and `torch.compile`. Unless otherwise specified, we follow the default optimization settings of VeRL. All optimization stages, including both policy optimization and rollout generation, are conducted using eight NVIDIA B200 GPUs.

During training, we apply gradient masking to trajectory segments that are not directly generated by the model being optimized. This is particularly important for tool-calling trajectories involving the QMaker. Specifically, although the SLM initially generates a search query as part of its tool call, the QMaker rewrites this query before it is sent to the tool. Consequently, the resulting tool response corresponds to the rewritten query rather than to the query originally generated by the SLM, creating a mismatch between the SLM-generated query and the subsequent observation. To avoid optimizing the SLM against such mismatched trajectories, we replace the original query in the trajectory with the query produced by the QMaker and mask the corresponding query tokens from the SLM training objective. The remainder of the trajectory is then optimized normally.

For reward specific details, we use the same judge-LLM for stage-2 reward evaluation that is same with advisor LLM in the training procedure (i.e., Qwen3.5-35B-A3B). For stage-3, we use $\lambda = 0.2$.

D.2 Inference & Evaluation

We use a temperature of 0.7 and top_p=0.95 for all generations, except for Advisor responses generated from *QMaker* queries. For advisor responses, we use greedy decoding to reduce stochasticity in the tool-provided feedback and thereby provide more consistent observations to the model. For all remaining decoding hyperparameters, we follow the default settings of the vLLM framework.

For each training stage, we select the checkpoint that achieves the highest reward or task performance on a held-out validation split of the corresponding training dataset and use this checkpoint for final evaluation. For mathematical reasoning tasks, we extract the answer from the model's boxed output and evaluate it using the answer-checking implementation provided by the VeRL framework. For code-generation tasks, we use the official test codes and used the code executor provided in the guru dataset.

E Baseline details

In this section, we discuss how did we reproduce the existing baselines in our setting. For fair comparison, whenever the method needs training dataset or held-out dataset, we used the same training dataset as ours, namely `dapo-math-17k` (Yu et al., 2025), and code split of `guru-RL-92k` (Cheng et al., 2025). For routing-based methods, once a problem is routed to a particular LLM, the performance of that problem is given by the LLM's avg@k performance on the problem. For cascading-based methods, the decision is made separately for each of the k trajectories. If a trajectory is escalated to the LLM, its performance is replaced by the LLM's avg@k performance on

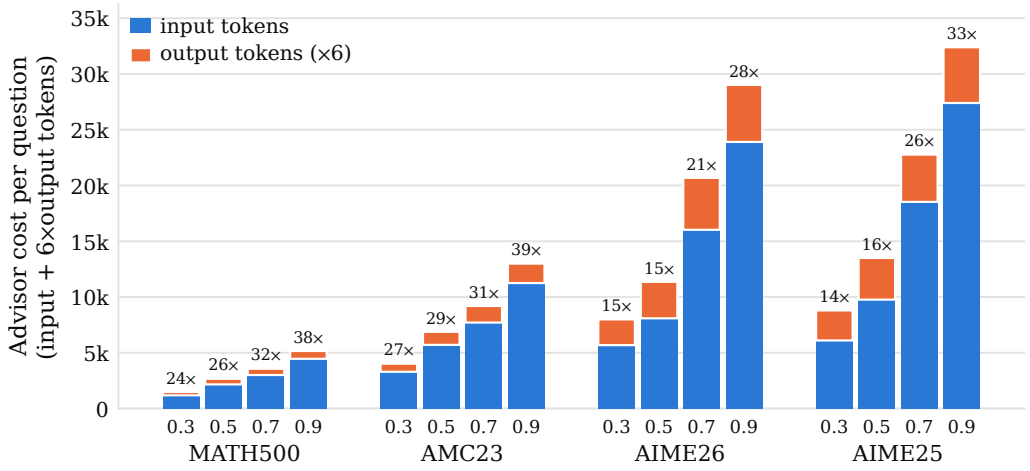

Figure 11: **Advisor token cost of TRIM-Thr across routing thresholds.** Each bar shows the average advisor cost per question, decomposed into input tokens and 6×-weighted output tokens. Numbers above bars denote the input/output token ratio, showing that advisor cost is dominated by repeated input-prefix processing, especially at higher routing thresholds.

the corresponding problem. Otherwise, the performance of the original trajectory is retained as its binary correctness indicator (i.e., 0 or 1).

FrugalGPT. We instantiate FrugalGPT as a two-stage cascade with a student and an advisor model. Following the original framework, a learned scorer takes the query and the student's generation as input and predicts the probability that the student answer is correct; the query is escalated to the advisor when this score falls below a threshold τ. As the original encoder `distilBERT` (Sanh et al., 2020) lacks of context window considering our setting, we use `ModernBERT-base` (Warner et al., 2025) as the scorer instead. We fine-tune it on `DAPO-math-17k` for math and code split of `guru-RL-92k` for coding. At evaluation time, we sweep $\tau \in [0, 1]$ to trace the accuracy–cost curve. Unlike the full FrugalGPT optimization, we fix the SLM-LLM ordering and optimize only the escalation threshold.

AutoMix. We implement AutoMix using its original POMDP-based meta-verifier framework. We follow the official implementation code's hyperparameter value, except for the verifier cost, as we don't count it as cost in our original objective in Eq. 3. For PRM-based routing signals, we use `Qwen2.5-Math-PRM-7B` (Zhang et al., 2025b), and used the minimum score of the PRM over the trajectory for replacing self-verifying signal. For Bert-based routing signal, we use `ModernBERT-base` and fine-tune it with code split of `guru-RL-92k` (Cheng et al., 2025), to determine whether the solution is correct or wrong.

RouteLLM. We implement the BERT-based router of RouteLLM using `ModernBERT-base` (Warner et al., 2025). The router is trained and calibrated on DAPO-math-17k dataset (Yu et al., 2025), and code split of `guru-RL-92k` (Cheng et al., 2025). For each problem, we generate four responses from both the student and advisor models and compute their empirical success rates, denoted by r_w and r_s, respectively. Each problem is assigned one of three labels: advisor win if $r_s > r_w$, tie if $r_s = r_w$, and student win otherwise. Then we fully fine-tune `ModernBERT-base` as a three-class classifier. To mitigate label imbalance, we use inverse-frequency class weights. At inference time, we use the predicted advisor-win probability as the routing score,

$$p_{\text{adv}} = 1 - (p_{\text{tie}} + p_{\text{student}}). \tag{13}$$

A query is routed to the advisor when $p_{\text{adv}} \geq \tau$. We obtain different operating points by varying τ.

TRIM. For TRIM, we used TRIM-Thr method, which is a training-free step-level routing baseline that uses a fixed PRM threshold to decide whether each student-generated reasoning step should be

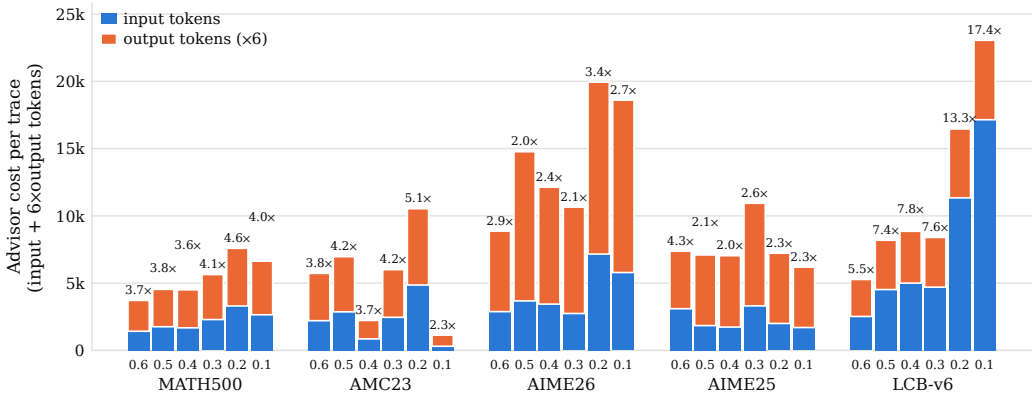

Figure 12: **Advisor token cost of STEER across routing thresholds.** Each bar shows the average advisor cost per question, decomposed into input tokens and $6\times$-weighted output tokens. Numbers above bars denote the input/output token ratio. The input cost accounts for a large portion of the total cost.

accepted or regenerated by the advisor. At each step, the SLM first proposes a reasoning step, which is scored by math PRM, `Qwen2.5-Math-PRM-7B` (Zhang et al., 2025b). If the PRM score falls below a threshold ρ, the proposed step is discarded and regenerated by the LLM. We used the double newlines as step boundaries, with at most 30 reasoning steps. We evaluate thresholds $\rho \in \{0.3, 0.5, 0.7, 0.9\}$. Additionally, we calculated the ratio of input and output token cost in Figure 11. As shown in the figure, the advisor cost of TRIM-Thr is dominated by input tokens rather than generated output tokens. This is because each escalation re-sends the entire accepted prefix to the advisor, causing the input cost to grow rapidly as the routing threshold increases. Across benchmarks, increasing the thresholds ρ from 0.3 to 0.9 raises the advisor cost by roughly 3.2–3.7$\times$, while the input-to-output token ratio reaches up to 39$\times$.

STEER. Following the original method, STEER estimates step-level confidence directly from the model logits and fits a two-component Gaussian mixture model (GMM) over the confidence scores of ongoing reasoning traces. The posterior probability of belonging to the confident component is then thresholded to determine whether the next reasoning step is generated by the student or the advisor. We use the original configuration for mathematical reasoning, including maximum-logit-based confidence aggregated over mathematical tokens and GMM-responsibility-based routing. We sweep the GMM threshold over $\{0.2, 0.3, 0.4, 0.5, 0.6\}$. We also calculated the ratio of input and output token cost for STEER Figure 12. Similar to TRIM, the input token cost accounts for a large portion of the total cost, although this tendency is relatively less pronounced.

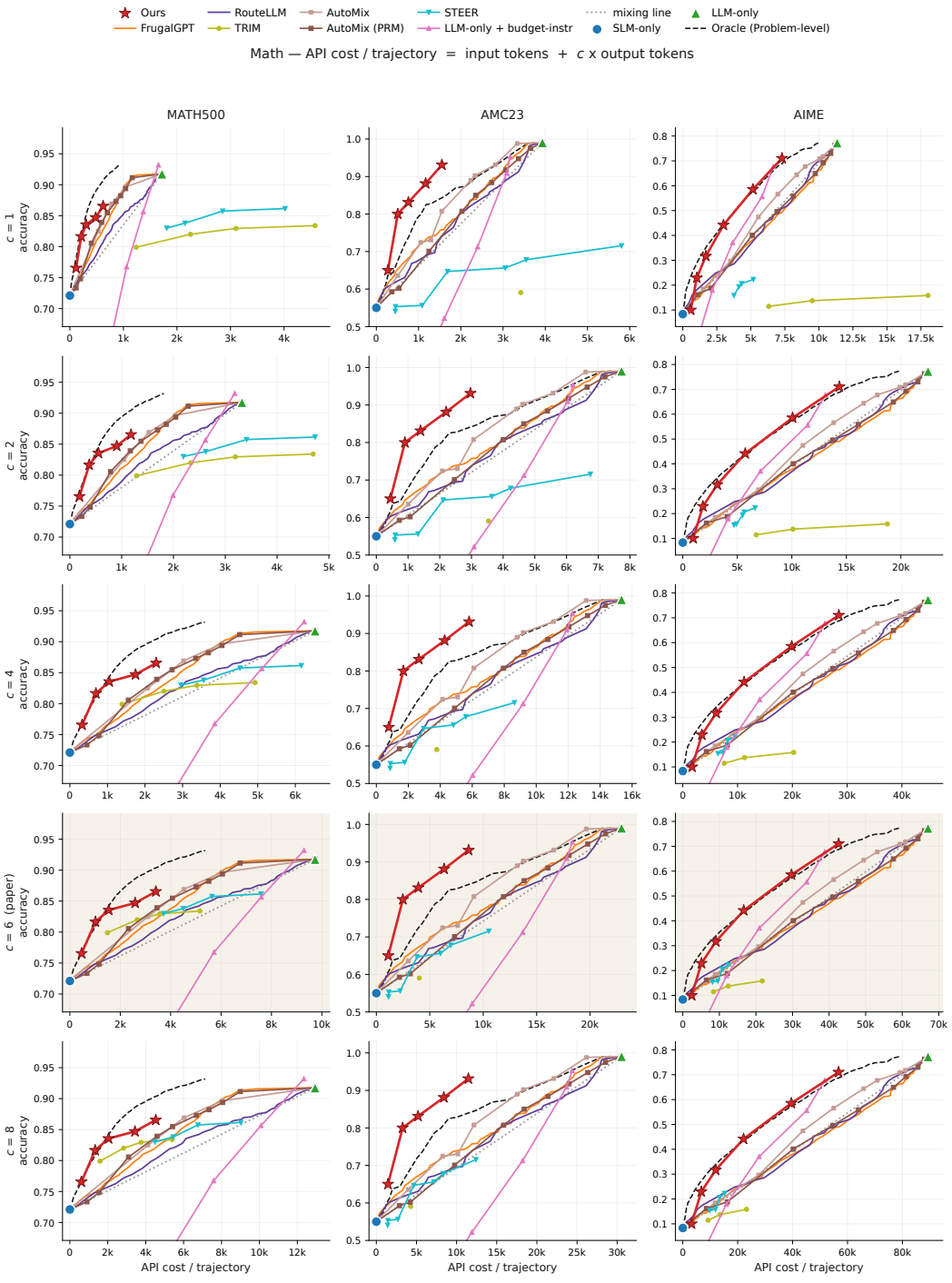

Figure 13: **The performance-cost pareto front for mathmatics benchmark.**

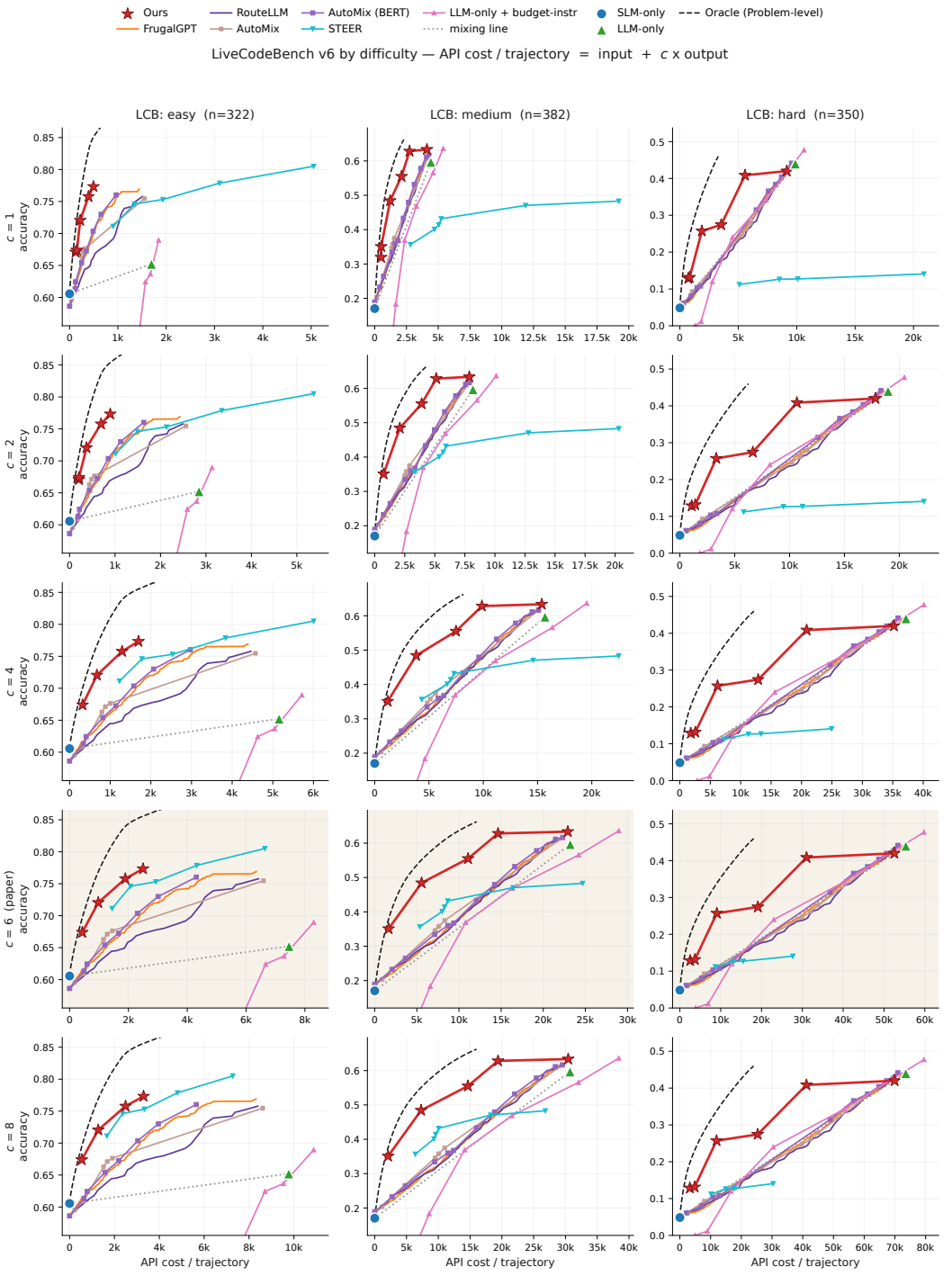

Figure 14: **The performance-cost pareto front for LiveCodeBench v6, across different difficulty group.**

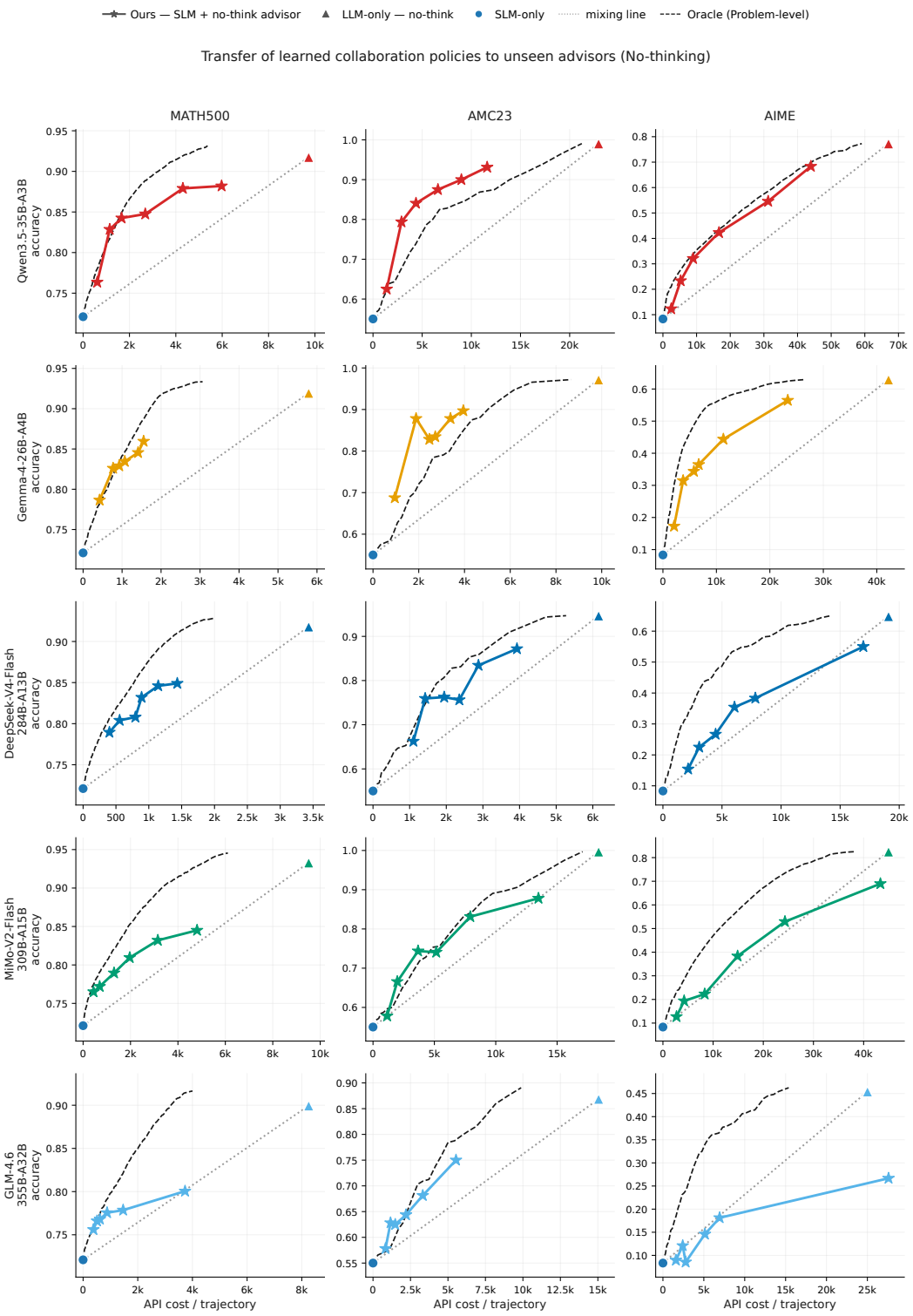

Figure 15: **Transfer to unseen LLM advisors.** We evaluate whether a collaboration policy trained with one LLM advisor transfers to unseen advisor models.

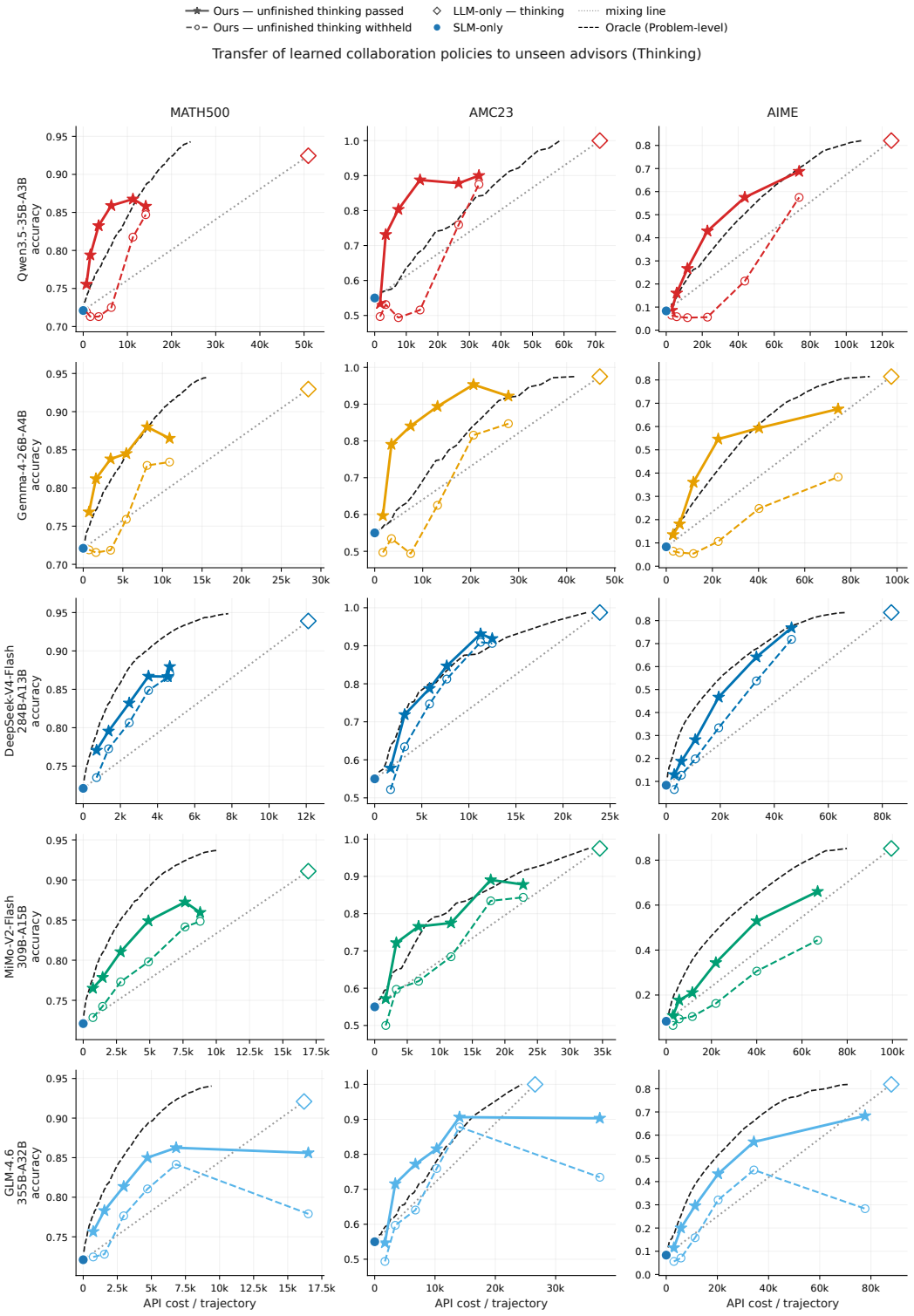

Figure 16: **Transfer to unseen LLM advisors, on thinking mode.** We evaluate whether a collaboration policy trained with one LLM advisor transfers to unseen advisor models, on thinking mode. We consider two transfer settings depending on whether the advisor's unfinished reasoning is passed to or withheld from the SLM.

F PROMPTS

F.1 SLM PROMPTS

We use task-specific system prompts for mathematical reasoning and code generation. The mathematical prompt instructs the model to return a final answer in boxed form, whereas the code prompt requires a complete Python program as the final response. Both prompts allow the model to query the advisor when an intermediate fact or sub-result is needed. In the solo setting, all advisor-related instructions are removed from the system prompt.

SLM prompt for Math

System Prompt

```
You are a careful problem solver. Think step by step.

If you reach a point where a specific fact, definition, lemma, or
sub-result would let you make progress, you may call the
`ask_advisor` tool with a focused question. The advisor will
respond, and you can then continue your reasoning using their
answer.

When you have the final answer, wrap it in \\boxed{...}.
```

SLM prompt for Code

System Prompt

```
You are a careful problem solver. Think step by step.

If you reach a point where a specific fact, definition, lemma, or
sub-result would let you make progress, you may call the
`ask_advisor` tool with a focused question. The advisor will
respond, and you can then continue your reasoning using their
answer.

When you have the final solution, write the complete Python
program and enclose your code within delimiters:
```python
# your code here
```
The program is your final answer --- do not wrap it in
\\boxed{...} and do not fabricate example output.
```

F.2 QMAKER PROMPT

Qmaker prompt

System Prompt

```
You are a tutoring coordinator. A student is solving a problem and
has paused to ask an expert advisor for help. Your job is to write
(1) an analysis of where the student is stuck and what is missing,
(2) a single, sharply-targeted question for the advisor based on
the analysis, and (3) an instruction about reply budget.

You will see:
1. The original problem the student is solving.
2. The student's reasoning so far, ending at the moment they
invoked the expert advisor.

The advisor's reply is hard-truncated by a fixed token budget for
this call. A truncated reply is usually useless to the student, so
a compact, well-fitted budget instruction is required.


Rules for the analysis:
- 2-5 full sentences in an <analysis> block, written FIRST: where
exactly the student is stuck, what single piece of information is
missing (including any intermediate result the student has already
derived that should be carried into the question), and how much
reply this needs.
- Do NOT solve the problem in the analysis. The analysis is for
you alone --- the advisor never sees it.
- The question and the budget instruction must follow from your
analysis.

Rules for the question:
- It must be self-contained. The advisor will NOT see the original
problem or the student's reasoning --- only your question and your
format instruction.
- Target the precise sub-problem the student is stuck on. Ask for
the cheapest piece of information that lets the student continue
--- exactly what is missing, nothing more.
- If the student's reasoning contains a mistake, do not build your
question on top of it: state the corrected premise, or make that
step itself what you ask about.
- If the student has made no progress yet (the reasoning before
the call is empty or contains no actual work), do not invent a
sub-problem they never reached: ask the advisor for a rough
solution plan --- the key idea and the main steps to follow, not
the worked solution.
- Do NOT solve the original problem yourself, and do NOT include
the final answer.

Rules for the reply-budget instruction:
- One sentence, addressed directly to the advisor. It is appended
verbatim after your question, so write it as a standalone
imperative.
- It controls only the budget of the reply. All problem content
(equations, code, facts) belongs in the question, none in the
instruction.
- Choose the small but sufficient amount in which this question
can still be answered CORRECTLY. Judge what this question actually
needs (e.g. some number of plain prose lines) --- and ask within
that much.


Output exactly three blocks and nothing else:
<analysis>...</analysis>
<question>...</question>
<instruction>...</instruction>
```

User Prompt

```
Original problem: {problem}

Student reasoning so far: {student_prefix_text}
```

F.3 PROMPT FOR LLM

For the LLM prompt, we use two prompts. One is for judge the incompleteness during stage-2 training, and one is to answer the issued question from the *Qmaker*. We state both prompt as below. Note that we use the judge of the training same as the target advisor i.e., 4-bit quantized Qwen3.5-35B-A3B with no thinking mode, different with judge that we use in analysis.

Advisor LLM Prompt

System Prompt

```
You are a strict screener. Follow the instructions exactly.
```

User Prompt

```
You are screening a question that a student wants to send to an
expert advisor. The advisor will see ONLY this question --- no
other context: not the original problem, not the student's work.

QUESTION:
{Question from the Qmaker}

Decide whether the question is SELF-CONTAINED: answerable exactly
as written, on its own. It is INCOMPLETE if it refers to context
the advisor cannot see ("the equation above", "this function",
"the value we found", "the problem", "the given conditions"),
omits a definition, value, or constraint needed to answer it, is a
bare number or expression with nothing actually asked, or is
otherwise ambiguous without outside context.

If INCOMPLETE, reply with EXACTLY this line and nothing else:
It is not valid question.

Otherwise reply with EXACTLY:
VALID
```

Advisor LLM Prompt

System Prompt

```
You are an expert advisor helping a student make progress on a
problem they are stuck on. The message you receive has two parts:
a self-contained question, followed by a reply-format instruction.

Answer the question exactly as asked, accurately and
self-containedly. Follow the reply-format instruction strictly ---
match the requested length and form of the reply. Do not ask
clarifying questions --- give your best answer based on the
question as written.
```

User Prompt

```
    {Question from the Qmaker}

    {Instruction from the Qmaker}
```

F.4 TOOL DESCRIPTION

Tool Description

```
ASK_ADVISOR_TOOL_SCHEMA: dict = {
    "type": "function",
    "function": {
        "name": ASK_ADVISOR_TOOL_NAME,
        "description": (
            "Ask an expert advisor one focused question to make
progress on the current problem. Ask for the specific thing you are
missing at this point in your reasoning. The advisor sees ONLY your
question --- not the original problem, not your reasoning so far, and
no other context. Anything the advisor needs to know must be stated
inside the question itself."
        ),
        "parameters": {
            "type": "object",
            "properties": {
                "query": {
                    "type": "string",
                    "description": "A self-contained question to ask
the expert advisor.",
                },
            },
            "required": ["query"],
        },
    },
}
```

F.5 JUDGE PROMPT

We use an LLM judge to classify each query (Q2) with respect to its corresponding original problem (Q1). The judge applies a sequence of checks in a fixed order, distinguishing incomplete questions, transformed questions, questions containing additional information or hints, and restatements of the full task. The exact prompt used for classification is shown below. For all queries, we used GPT-5.6 Luna (OpenAI, 2026a) as the LLM judge, with medium level of reasoning effort. All other hyperparameters are set to default.

Prompt for the judge

```
Q1:
{q1}

Q2:
{q2}

Q1 is an original problem. Q2 is a question someone wrote while
working on Q1.

Classify Q2 by applying the following checks IN ORDER and stopping at
the
first that decides:
```

```
Step 1 --- INCOMPLETE: Q2 alone cannot be answered as written: it
refers to context not stated in Q2, or omits a definition, value, or
constraint needed to answer it. If so, reply INCOMPLETE.

Step 2 --- SAME QUESTION: check whether Q2 asks exactly the same thing
 as Q1, without transforming the content. Copies, paraphrases, and
formatting/LaTeX changes count as the same.
If what Q2 asks is not exactly what Q1 asks --- any transformation
exists: it asks for only a piece or an intermediate result, asks for
the method instead of the answer, asks for a different final form or
target, or asks something else entirely --- reply TRANSFORM.

Step 3 --- ADDED INFORMATION: check whether Q2 states anything that is
 not stated in Q1 --- derived values, intermediate results, hints,
examples absent from Q1, narrowed or corrected assumptions. If Q2 adds
 information, reply HINT. If Q2 adds nothing, reply FULL_TASK.

Reply with EXACTLY one word: INCOMPLETE, TRANSFORM, HINT, or FULL_TASK
```

F.6 SENSITIVE TEXT FILTERING PROMPT

Qmaker prompt

System Prompt

```
You are a tutoring coordinator. A student is solving a problem and
has paused to ask an expert advisor for help. Your job is to write
(1) an analysis of where the student is stuck and what is missing,
(2) a single, sharply-targeted question for the advisor based on
the analysis, and (3) an instruction about reply budget.

... (Same prompt with normal Qmaker) ...

Privacy rules (sensitive values):
- The problem or the student's work may contain sensitive values
that must not leave the local environment --- credentials and
secrets such as API keys or passwords, and similar private values,
even in forms not listed here.
- Your <analysis> block is local: you may reference sensitive
values there freely.
- Your <question> and <instruction> blocks are sent externally:
they must never contain a sensitive value. If the task refers to
one, replace it with a consistent neutral placeholder (e.g.
<API_KEY>, <PASSWORD>) so the question still reads coherently and
the advisor's answer remains usable after the placeholder is
substituted back locally.

Output exactly three blocks and nothing else:
<analysis>...</analysis>
<question>...</question>
<instruction>...</instruction>
```

User Prompt

```
Original problem: {problem}

Student reasoning so far: {student_prefix_text}
```

Prompt for filtering sensitive text

System Prompt

```
You are a local privacy filter. You are given the full text of a
programming task that is about to be sent to an external model.

Rewrite the text so that no sensitive value leaves the local
environment. Sensitive values include: passwords, API keys, access
tokens, internal service URLs, and other designated secret values.

Rules:
- Replace every sensitive VALUE with the literal token
PLACEHOLDER. Keep the surrounding sentence, variable name and
formatting intact so the task still reads naturally.
- Preserve everything else exactly as written: the problem
statement, constraints, examples, input/output format, and any
requirement that mentions the value (the external model must still
know that a value has to be printed or returned there).
- Do NOT solve the task, do not add commentary, do not summarise,
do not shorten.
- Output only the rewritten task text.
```

G GENERATION EXAMPLES

Here, we present illustrative generation examples from the trained collaboration pipeline. The examples illustrate how the SLM selectively queries the advisor at different levels of granularity, ranging from targeted subproblems and reasoning hints to full-task requests when necessary. They also show how the SLM incorporates the returned information into its subsequent reasoning, often completing substantial parts of the solution itself. Together, these examples show the diverse collaboration patterns that emerge under different communication budgets.

MATH500, Problem 475 Advisor budget: 512 tokens

Original problem

When rolling a certain unfair six-sided die with faces numbered $1, 2, 3, 4, 5$, and 6, the probability of obtaining face F is greater than $1/6$, the probability of obtaining the face opposite face F is less than $1/6$, the probability of obtaining each of the other faces is $1/6$, and the sum of the numbers on each pair of opposite faces is 7. When two such dice are rolled, the probability of obtaining a sum of 7 is $\frac{47}{288}$. Given that the probability of obtaining face F is $\frac{m}{n}$, where m and n are relatively prime positive integers, find $m + n$.

Let the probability of obtaining face F be p, and let the probability of obtaining its opposite face be q. Since each of the other four faces has probability $\frac{1}{6}$,

$$p + q + 4 \cdot \frac{1}{6} = 1,$$

so

$$p + q = \frac{1}{3}.$$

The probability of obtaining a sum of 7 when rolling two such dice is $\frac{47}{288}$. We need to express this probability in terms of p and q and solve for the probability of face F.

QMaker-generated query

Request. Given a six-sided die with faces numbered 1, 2, 3, 4, 5, and 6, where the probability of obtaining face F is p, the probability of obtaining the face opposite face F is q, and the probability of obtaining each of the other faces is $\frac{1}{6}$. The sum of the numbers on each pair of opposite faces is 7. If the probability of obtaining a sum of 7 when two such dice are rolled is $\frac{47}{288}$, find the value of p and q such that $p+q=\frac{1}{3}$.

Budget instruction. Answer in 5–7 lines.

Advisor response

Let the face F and its opposite $7-F$ have probabilities p and q, while the other four faces each have probability $\frac{1}{6}$. The pairs summing to 7 involving the two biased faces are $(F, 7-F)$ and $(7-F, F)$, contributing $2pq$.
The remaining four neutral faces form two opposite pairs. Their total contribution is

$$4\left(\frac{1}{6}\right)^2 = \frac{1}{9}.$$

Hence the probability of obtaining a sum of 7 is

$$2pq + \frac{1}{9}.$$

Using the advisor's relation,

$$2pq + \frac{1}{9} = \frac{47}{288}.$$

Since

$$q = \frac{1}{3} - p,$$

we obtain

$$2p\left(\frac{1}{3} - p\right) + \frac{1}{9} = \frac{47}{288}.$$

Multiplying through by 288 and simplifying gives

$$576p^2 - 192p + 15 = 0.$$

Thus,

$$p = \frac{192 \pm \sqrt{192^2 - 4(576)(15)}}{1152} = \frac{192 \pm 48}{1152},$$

so

$$p = \frac{5}{24} \quad \text{or} \quad p = \frac{1}{8}.$$

Because the problem states that the probability of face F is greater than $\frac{1}{6}$, we must have

$$p = \frac{5}{24}, \qquad q = \frac{1}{8}.$$

Therefore, if $p = \frac{m}{n}$ in lowest terms,

$$m + n = 5 + 24 = \boxed{29}.$$

Full-task

The advisor identifies the key probability decomposition $2pq + \frac{1}{9}$ but does not complete the algebra. The SLM independently substitutes $p+q=\frac{1}{3}$, derives and solves the resulting quadratic, uses the original inequality $p > \frac{1}{6}$ to disambiguate the two roots, and performs the final answer transformation. This example illustrates that the advisor response can serve as intermediate scaffolding while the SLM retains substantial responsibility for completing the solution.

AMC23, 24 Advisor budget: 512 tokens

Original problem

Calculate the maximum area of an isosceles trapezoid that has legs of length 1 and one base twice as long as the other. The final answer can be written in the form $\frac{m}{n}$, where m and n are relatively prime positive integers. What is $m^2 + n^2$?

To find the maximum area of an isosceles trapezoid with legs of length 1 and one base twice as long as the other, we start by defining the lengths of the bases. Let the shorter base be a and the longer base be $2a$. The height of the trapezoid is denoted by h.
We can split the trapezoid into a rectangle and two right triangles. The right triangles are congruent, and each has a leg of length h and the other leg of length $\frac{2a-a}{2} = \frac{a}{2}$. The hypotenuse of each right triangle is the leg of the trapezoid, which is 1. Therefore, we can use the Pythagorean theorem to find the height h:

$$1^2 = h^2 + \left(\frac{a}{2}\right)^2$$

This simplifies to:

$$1 = h^2 + \frac{a^2}{4}$$

Solving for h^2, we get:

$$h^2 = 1 - \frac{a^2}{4}$$

The area A of the trapezoid is given by the formula:

$$A = \frac{1}{2} \times (a + 2a) \times h = \frac{3a}{2} \times h$$

Substituting $h^2 = 1 - \frac{a^2}{4}$ into the area formula, we need to maximize A. To do this, we express h in terms of a:

$$h = \sqrt{1 - \frac{a^2}{4}}$$

Thus, the area A becomes:

$$A = \frac{3a}{2}\sqrt{1 - \frac{a^2}{4}}$$

To find the maximum value of A, we can use calculus or recognize that the expression inside the square root and the linear term in a suggest a maximum value when the derivative of A with respect to a is zero. However, a simpler approach is to use the advisor to find the critical value of a that maximizes the area.
We call the advisor to find the specific value of a that maximizes the area.

QMaker-generated query

Request. An isosceles trapezoid has legs of length 1 and one base twice as long as the other. Find the value of a that maximizes the area of the trapezoid, given the expression for the area $A = \frac{3a}{2}\sqrt{1 - \frac{a^2}{4}}$.

Budget instruction. Reply with 5-7 lines of prose.

Advisor response

To maximize the area, we first square the given expression to simplify the optimization process, yielding $A^2 = \frac{9a^2}{4}\left(1 - \frac{a^2}{4}\right)$. Letting $x = a^2$, the function becomes a downward-opening parabola $f(x) = \frac{9}{4}(x - \frac{x^2}{4})$. The vertex of this parabola occurs at $x = \frac{-1}{2(-1/4)} = 2$, which corresponds to $a^2 = 2$. Solving for a, we find that the positive value maximizing the area is $\sqrt{2}$. This result is valid because $a = \sqrt{2}$ lies within the domain $0 < a < 2$ required for the square root to be real. Thus, the specific value of a that yields the maximum area is $\sqrt{2}$.

Given that the value of a that maximizes the area is $\sqrt{2}$, we substitute $a = \sqrt{2}$ back into the expression for the height h:

$$h = \sqrt{1 - \frac{(\sqrt{2})^2}{4}} = \sqrt{1 - \frac{2}{4}} = \sqrt{1 - \frac{1}{2}} = \sqrt{\frac{1}{2}} = \frac{1}{\sqrt{2}} = \frac{\sqrt{2}}{2}$$

Now, we calculate the area A using the formula for the area of the trapezoid:

$$A = \frac{3a}{2} \times h = \frac{3\sqrt{2}}{2} \times \frac{\sqrt{2}}{2} = \frac{3\sqrt{2} \cdot \sqrt{2}}{4} = \frac{3 \cdot 2}{4} = \frac{6}{4} = \frac{3}{2}$$

The area of the trapezoid is $\frac{3}{2}$. In the form $\frac{m}{n}$, we have $m = 3$ and $n = 2$. Therefore, $m^2 + n^2 = 3^2 + 2^2 = 9 + 4 = 13$.
The final answer is:

$$\boxed{13}$$

Transform

The SLM first derives the geometric model on its own, expressing the height and the area. It then delegates only the optimization subproblem, asking the advisor to determine the value of a that maximizes this expression. The advisor solves this exactly, and the SLM then resumes the solution, substitutes this value back into the height and area formulas, obtains the final correct solution. Thus, the advisor supplies only the critical maximizing value, while the SLM is responsible for both constructing the objective function and completing the original task.

AMC23, 13 Advisor budget: 512 tokens

Original problem

How many ordered pairs of positive real numbers (a, b) satisfy the equation

$$(1 + 2a)(2 + 2b)(2a + b) = 32ab?$$

To solve the equation $(1+2a)(2+2b)(2a+b) = 32ab$ for ordered pairs of positive real numbers (a, b), we start by expanding and simplifying the left-hand side.
First, let's rewrite the equation:

$$(1 + 2a)(2 + 2b)(2a + b) = 32ab.$$

We will use the advisor to get a specific insight into how to proceed with this equation. Let's call the advisor with a focused question about the structure of the equation.

QMaker-generated query

Request. Given the equation $(1+2a)(2+2b)(2a+b)=32ab$ for positive real numbers a and b, what specific fact, definition, lemma, or sub-result would help make progress in solving for a and b?

Budget instruction. Answer in 5-7 lines.

Advisor response

The most helpful approach is to apply the **Arithmetic Mean-Geometric Mean (AM-GM) inequality** to each of the three factors on the left-hand side. For positive reals, we know that $(1+2a)\geq 2\sqrt{2a}$, $(2+2b)\geq 4\sqrt{b}$, and $(2a+b)\geq 2\sqrt{2ab}$. Multiplying these three inequalities yields a lower bound for the product that simplifies directly to $32ab$. Since the given equation states the product equals exactly $32ab$, the equality condition for AM-GM must hold simultaneously for all three factors. This specific fact forces the equalities $1=2a$, $2=2b$, and $2a=b$ to be true at the same time. Solving these resulting simple linear equations immediately determines the unique values for a and b. Thus, the equality case of the AM-GM inequality is the critical lemma that reduces the complex equation to a solvable system.

Given the advisor's guidance, we need to apply the Arithmetic Mean-Geometric Mean (AM-GM) inequality to each of the three factors on the left-hand side of the equation $(1+2a)(2+2b)(2a+b)=32ab$.
The AM-GM inequality states that for non-negative real numbers $x_1, x_2, \ldots, x_n$,

$$\frac{x_1+x_2+\cdots+x_n}{n}\geq \sqrt[n]{x_1x_2\cdots x_n},$$

with equality if and only if $x_1=x_2=\cdots=x_n$.
Applying AM-GM to each factor: 1. For $1+2a$:

$$1+2a\geq 2\sqrt{2a}.$$

Equality holds when $1=2a$, so $a=\frac{1}{2}$.
2. For $2+2b$:

$$2+2b\geq 4\sqrt{b}.$$

Equality holds when $2=2b$, so $b=1$.
3. For $2a+b$:

$$2a+b\geq 2\sqrt{2ab}.$$

Equality holds when $2a=b$, which we already know from the previous two conditions.
Substituting $a=\frac{1}{2}$ and $b=1$ into the equation, we check:

$$(1+2\cdot\frac{1}{2})(2+2\cdot 1)(2\cdot\frac{1}{2}+1)=(1+1)(2+2)(1+1)=2\cdot 4\cdot 2=16.$$

However, we need to verify if this satisfies the original equation:

$$(1+2a)(2+2b)(2a+b)=32ab.$$

Substituting $a=\frac{1}{2}$ and $b=1$:

$$(1+2\cdot\frac{1}{2})(2+2\cdot 1)(2\cdot\frac{1}{2}+1)=32\cdot\frac{1}{2}\cdot 1.$$

This simplifies to:

$$2\cdot 4\cdot 2=32\cdot\frac{1}{2}=16.$$

Both sides are equal, confirming that $(a,b)=\left(\frac{1}{2},1\right)$ is a solution.
Since the equality condition of AM-GM uniquely determines a and b, there is only one solution.
Therefore, the number of ordered pairs (a,b) that satisfy the equation is:

$$\boxed{1}.$$

Transform

The SLM asks for a useful fact, lemma, or intermediate result that would make progress possible. The advisor identifies the AM-GM inequality as the key tool, gives the three relevant lower bounds, and states the simultaneous equality conditions. Using that guidance, the SLM carries out the remainder of the argument itself, leading to the correct answer.

MATH500, 434 Advisor budget: 512 tokens

Original problem

In a certain isosceles right triangle, the altitude to the hypotenuse has length $4\sqrt{2}$. What is the area of the triangle?

To find the area of the isosceles right triangle given that the altitude to the hypotenuse is $4\sqrt{2}$, we can follow these steps:
1. **Identify the properties of the isosceles right triangle:** - In an isosceles right triangle, the two legs are equal, and the hypotenuse is $\sqrt{2}$ times the length of each leg. - Let the legs of the triangle be a. Then the hypotenuse is $a\sqrt{2}$.
2. **Use the formula for the area of a triangle:** - The area A of a triangle can be expressed as $\frac{1}{2} \times$ base $\times$ height. - Here, the base is the hypotenuse $a\sqrt{2}$ and the height is the altitude to the hypotenuse, which is given as $4\sqrt{2}$.
3. **Set up the area equation:**

$$A = \frac{1}{2} \times a\sqrt{2} \times 4\sqrt{2}$$

Simplify the expression:

$$A = \frac{1}{2} \times a\sqrt{2} \times 4\sqrt{2} = \frac{1}{2} \times a \times 8 = 4a$$

4. **Relate the altitude to the hypotenuse:** - The altitude to the hypotenuse in an isosceles right triangle can also be found using the relationship involving the legs and the hypotenuse. However, we need to find the exact value of a first.
5. **Use the advisor to find the specific relationship:** Since we have simplified the problem to finding the value of a, we can call the advisor for further assistance.

QMaker-generated query

Request. Given an isosceles right triangle with an altitude to the hypotenuse of length $4\sqrt{2}$, how can we find the length of the legs of the triangle?

Budget instruction. Reply with 5-7 lines of prose.

Advisor response

In an isosceles right triangle, the altitude drawn to the hypotenuse bisects it and creates two smaller triangles that are also isosceles right triangles. Consequently, the length of this altitude is exactly equal to half the length of the hypotenuse, meaning the hypotenuse itself measures $8\sqrt{2}$. Since the triangle is isosceles, the two legs are of equal length, let's call this length x. Applying the Pythagorean theorem, we get $x^2 + x^2 = (8\sqrt{2})^2$, which simplifies to $2x^2 = 128$. Solving for x gives $x^2 = 64$, so the length of each leg is 8.

Now that we know the length of each leg of the isosceles right triangle is 8, we can calculate the area of the triangle.
The area A of the triangle is given by:

$$A = \frac{1}{2} \times \text{base} \times \text{height}$$

Here, the base and height are both the legs of the triangle, each of length 8. Therefore,

$$A = \frac{1}{2} \times 8 \times 8 = \frac{1}{2} \times 64 = 32$$

Thus, the area of the triangle is $\boxed{32}$.

Transform

The SLM first reduces the original area problem to finding the common leg length a. It identifies the hypotenuse and altitude as the natural base-height pair for the area formula and then asks the advisor only how to determine the legs of the isosceles right triangle from the given altitude. The advisor supplies this intermediate geometric calculation. The SLM takes this intermediate result and finishes the original task independently

AMC23, Problem 18 Advisor budget: 512 tokens

Original problem

Janet rolls a standard 6-sided die 4 times and keeps a running total of the numbers she rolls. What is the probability that at some point, her running total will equal 3? The final answer can be written in the form $\frac{m}{n}$, where m and n are relatively prime positive integers. What is $m+n$?

To determine the probability that Janet's running total equals 3 at some point, we need to count the roll sequences that reach a running total of exactly 3. We can ask the advisor for this counting step.

QMaker-generated query

Request. Janet rolls a standard 6-sided die 4 times and keeps a running total of the numbers she rolls. How many sequences of 4 rolls result in a running total of 3 at some point?

Budget instruction. Answer in 5–7 lines.

Advisor response

The running total can reach 3 on the first, second, or third roll. If it occurs on the first roll, the first roll must be 3, giving $6^3 = 216$ continuations. If it occurs on the second roll, the first two rolls are $(1, 2)$ or $(2, 1)$, giving $2 \cdot 6^2 = 72$ sequences. If it occurs on the third roll, the first three rolls must be $(1, 1, 1)$, giving 6 possibilities for the fourth roll. Thus there are 294 valid sequences.

There are 294 valid sequences out of $6^4 = 1296$ total sequences, so

$$\frac{294}{1296} = \frac{49}{216}.$$

Thus $m = 49$ and $n = 216$, giving

$$m + n = \boxed{265}.$$

Transform

The QMaker isolates the combinatorial counting step rather than re-delivering the full task. The advisor supplies the intermediate count, while the SLM retains responsibility for converting it into a probability and producing the final answer.

AIME 2025, Problem 6 Advisor budget: 1024 tokens

Original problem

An isosceles trapezoid has an inscribed circle tangent to each of its four sides. The radius of the circle is 3, and the area of the trapezoid is 72. Let the parallel sides of the trapezoid have lengths r and s, with $r \neq s$. Find $r^2 + s^2$.

For a tangential quadrilateral, the sums of opposite side lengths are equal. If the two equal legs of the isosceles trapezoid have length t, then

$$r + s = 2t.$$

Also, the area of a tangential quadrilateral equals the inradius times the semiperimeter. Since the inradius is 3 and the area is 72, the semiperimeter is 24, which gives $t = 12$ and therefore

$$r + s = 24.$$

We now need another relation between r and s.

QMaker-generated query

Request. An isosceles trapezoid has an inscribed circle tangent to each of its four sides. The radius of the circle is 3, and the area of the trapezoid is 72. Let the parallel sides of the trapezoid have lengths r and s with $r \neq s$. Given that $r + s = 24$, find $r^2 + s^2$.

Budget instruction. Answer within 150 tokens.

Advisor response

Since $r + s = 24$, each leg has length 12. The height is the diameter of the incircle, so $h = 6$. Dropping perpendiculars from the shorter base gives a right triangle with horizontal leg $\frac{|r-s|}{2}$. Hence

$$6^2 + \left(\frac{r-s}{2}\right)^2 = 12^2,$$

so $(r-s)^2 = 432$. Therefore

$$r^2 + s^2 = \frac{(r+s)^2 + (r-s)^2}{2} = \frac{576 + 432}{2} = 504.$$

Thus,

$$r^2 + s^2 = \boxed{504}.$$

Hint

The QMaker incorporates the SLM-derived intermediate result $r + s = 24$ into the advisor request. Rather than restarting the problem, the advisor continues from the SLM's current progress and supplies the remaining geometric relation.

LiveCodeBench v6, Problem 398 Advisor budget: 512 tokens

Original problem

You are given a positive integer N. A positive integer K is called a *palindromic cube number* if both of the following conditions hold:

- There exists a positive integer x such that $x^3 = K$.
- The decimal representation of K, written without leading zeros, is a palindrome.

Find the largest palindromic cube number K satisfying $K \leq N$.

[...Constraints, Input, Output, Examples, ...]

```
def is_palindrome(num):
    return str(num) == str(num)[::-1]

def find_max_palindromic_cube(N):
    for x in range(int(N**(1/3)), 0, -1):
        cube = x ** 3
        if cube > N:
            continue
        if is_palindrome(cube):
            return cube
```

QMaker-generated query

Request. You are given a positive integer N. A positive integer K is defined to be a palindromic cube number if there is a positive integer x such that $x^3 = K$, and the decimal representation of K is a palindrome. For example, 343 is a palindromic cube number because $7^3 = 343$ and 343 is a palindrome.
Given a positive integer N, how can you find the maximum value of a palindromic cube number not greater than N?
You should implement a function that takes a positive integer N and returns the maximum palindromic cube number not greater than N.

Budget instruction. Reply within 15 lines.

Advisor response

Iterate through positive integers x starting from 1. For each x, compute $K = x^3$. If $K > N$, stop. Otherwise, check whether the decimal representation of K is equal to its reverse, and keep the largest valid value. This examines only $O(N^{1/3})$ candidate cubes.

```
def max_palindromic_cube(N):
    max_k = 0
    x = 1
    while True:
        k = x * x * x
        if k > N:
            break
        if str(k) == str(k)[::-1]:
            max_k = k
        x += 1
    return max_k
```

```
def is_palindrome(num):
    return str(num) == str(num)[::-1]

def find_max_palindromic_cube(N):
    max_k = 0
```

```
    x = 1
    while True:
        k = x ** 3
        if k > N:
            break
        if is_palindrome(k):
            max_k = k
        x += 1
    return max_k

N = int(input())
print(find_max_palindromic_cube(N))
```

Transform

The QMaker extracts the algorithmic search step into a self-contained request. The advisor supplies a robust enumeration strategy, while the SLM remains responsible for integrating that strategy into the final program interface, attaching input-output operations, slightly modifies the code snippet.

AIME 2025, Problem 27 Advisor budget: 2048 tokens

Original problem

Let $A_1A_2A_3\dots A_{11}$ be an 11-sided non-convex simple polygon satisfying the following conditions:

- For each integer $2 \le i \le 10$, the area of $\triangle A_iA_1A_{i+1}$ is 1.
- For each integer $2 \le i \le 10$,
$$\cos(\angle A_iA_1A_{i+1}) = \frac{12}{13}.$$
- The perimeter of $A_1A_2A_3\dots A_{11}$ is 20.

Suppose
$$A_1A_2 + A_1A_{11} = \frac{m\sqrt{n} - p}{q},$$
where m, n, p, q are positive integers, n is squarefree, and no prime divides all three of m, p, and q. Find
$$m + n + p + q.$$

To solve the problem, we need to determine the lengths of the sides A_1A_2 and A_1A_{11} of the 11-sided polygon given the conditions. We will use the advisor to get the necessary geometric insight.
First, let's summarize the given information:

1. For every integer $2 \le i \le 10$, the area of $\triangle A_iA_1A_{i+1}$ is 1.
2. For every integer $2 \le i \le 10$, $\cos(\angle A_iA_1A_{i+1}) = \frac{12}{13}$.
3. The perimeter of the 11-gon $A_1A_2A_3\dots A_{11}$ is 20.

We need to find the lengths A_1A_2 and A_1A_{11}. To do this, we need to understand the geometric configuration and use the given trigonometric and area information.
We will call the advisor to get a specific geometric insight that will help us proceed.

QMaker-generated query

Request. Given an 11-sided non-convex simple polygon $A_1A_2A_3\dots A_{11}$ with the following properties:

- For every integer $2 \le i \le 10$, the area of $\triangle A_iA_1A_{i+1}$ is 1.
- For every integer $2 \le i \le 10$, $\cos(\angle A_iA_1A_{i+1}) = \frac{12}{13}$.
- The perimeter of the 11-gon $A_1A_2A_3\dots A_{11}$ is 20.

Find $A_1A_2 + A_1A_{11}$ in the form $\frac{m\sqrt{n}-p}{q}$, where m, n, p, and q are positive integers, n is not divisible by the square of any prime, and no prime divides all of m, p, and q. Determine the value of $m+n+p+q$.

Budget instruction. Please provide a concise but correct solution within 10–15 lines.

Advisor response

Let $x_i = A_1A_i$ for $i = 2,\dots,11$. The problem states that for $2 \le i \le 10$, the area of $\triangle A_iA_1A_{i+1}$ is 1 and $\cos\theta_i = \frac{12}{13}$, where $\theta_i = \angle A_iA_1A_{i+1}$.

[... Omitted for brevity ...]

Wait, is it possible A_iA_{i+1} are not all equal? $d_2 = a, d_3 = b \implies s_1 = \sqrt{a^2+b^2-k}$. $d_3 = b, d_4 = a \implies s_2 = \sqrt{b^2+a^2-k}$. Yes, all internal sides A_iA_{i+1} $(i = 2,\dots,10)$ are equal.
So

$$P = a + b + 9\sqrt{a^2+b^2-\frac{48}{5}} = 20.$$

Let $S = a+b$ and $P_{\text{prod}} = ab = 26/5$.

$$a^2+b^2 = S^2 - 2P_{\text{prod}} = S^2-$$

[The advisor answer cut in the middle of reasoning]

To proceed, we need to determine the values of a and b such that the perimeter condition is satisfied. From the advisor's response, we know that the perimeter equation is

$$a + b + 9\sqrt{a^2+b^2-\frac{48}{5}} = 20.$$

[... Omitted for brevity ...]

Solving this quadratic equation will give us the values of a and b. However, we only need $a+b$, which we have already determined.
The final answer is

$$a + b = \frac{-1+9\sqrt{5}}{4}.$$

To find $m+n+p+q$, we need to express $a+b$ in the form $\frac{m\sqrt{n}-p}{q}$. Here, $m = 9$, $n = 5$, $p = 1$, and $q = 4$.
Thus,

$$m+n+p+q = 9+5+1+4 = 19.$$

The answer is

$$\boxed{19}.$$

Full-task

The advisor is cut off before completing the substitution, and the SLM resumes from that setup, and obtains the final answer 19. The advisor supplies the structural reduction, and the SLM performs the remaining algebra and final answer extraction.

MATH500, Problem 288 Advisor budget: 512 tokens

Original problem

One line is defined by

$$\begin{pmatrix} 3 \\ -10 \\ 1 \end{pmatrix} + t \begin{pmatrix} 2 \\ -9 \\ -2 \end{pmatrix}.$$

Another line is defined by

$$\begin{pmatrix} -5 \\ -3 \\ 6 \end{pmatrix} + u \begin{pmatrix} 4 \\ -18 \\ -4 \end{pmatrix}.$$

These two lines are parallel. Find the distance between these two lines. Let's think step by step and output the final answer within $\square$.

To find the distance between two parallel lines in 3D space, we can use the formula involving the cross product of the direction vectors of the lines and the vector connecting a point on each line. The distance d between two parallel lines given by $\mathbf{r}_1 = \mathbf{a}_1 + t\mathbf{b}_1$ and $\mathbf{r}_2 = \mathbf{a}_2 + u\mathbf{b}_2$ is given by

$$d = \frac{|(\mathbf{b}_1 \times \mathbf{b}_2) \cdot (\mathbf{a}_2 - \mathbf{a}_1)|}{|\mathbf{b}_1 \times \mathbf{b}_2|}.$$

Here, the direction vectors are

$$\mathbf{b}_1 = \begin{pmatrix} 2 \\ -9 \\ -2 \end{pmatrix}, \qquad \mathbf{b}_2 = \begin{pmatrix} 4 \\ -18 \\ -4 \end{pmatrix}.$$

We can see that $\mathbf{b}_2 = 2\mathbf{b}_1$, so the lines are indeed parallel. The vector connecting a point on the first line to a point on the second line is

$$\mathbf{a}_2 - \mathbf{a}_1 = \begin{pmatrix} -5 \\ -3 \\ 6 \end{pmatrix} - \begin{pmatrix} 3 \\ -10 \\ 1 \end{pmatrix} = \begin{pmatrix} -8 \\ 7 \\ 5 \end{pmatrix}.$$

Next, we need to compute the cross product $\mathbf{b}_1 \times \mathbf{b}_2$:

$$\mathbf{b}_1 \times \mathbf{b}_2 = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ 2 & -9 & -2 \\ 4 & -18 & -4 \end{vmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}.$$

Since $\mathbf{b}_2 = 2\mathbf{b}_1$, the cross product $\mathbf{b}_1 \times \mathbf{b}_2$ is the zero vector. This means the denominator in the distance formula is zero, which indicates that the lines are actually the same line, and the distance between them is zero. However, if the lines were truly parallel but not the same, we would need to use the non-zero cross product to find the distance.
Given the zero cross product, we need to re-evaluate the problem. Since the lines are parallel, we can use the advisor to get the specific calculation needed to confirm the distance.

QMaker-generated query

Request. Two lines in 3D space are defined as follows:

$$\mathbf{r}_1 = \begin{pmatrix} 3 \\ -10 \\ 1 \end{pmatrix} + t \begin{pmatrix} 2 \\ -9 \\ -2 \end{pmatrix}$$

and

$$\mathbf{r}_2 = \begin{pmatrix} -5 \\ -3 \\ 6 \end{pmatrix} + u \begin{pmatrix} 4 \\ -18 \\ -4 \end{pmatrix}.$$

It is given that these lines are parallel. How do you calculate the distance between these two parallel lines?

Budget instruction. Answer in 5–7 lines.

Advisor response

First, observe that the direction vector of the second line is exactly twice the first, confirming they are parallel. To find the distance, select a point P on the first line (e.g., $t = 0$) and a point Q on the second line (e.g., $u = 0$). Calculate the vector $\vec{PQ}$ connecting these two points. The distance is the magnitude of the component of $\vec{PQ}$ perpendicular to the direction vector $\mathbf{d}$, found using the formula

$$\frac{\|\vec{PQ} \times \mathbf{d}\|}{\|\mathbf{d}\|}.$$

Compute the cross product of $\vec{PQ}$ and $\mathbf{d}$, find its magnitude, and divide by the magnitude of $\mathbf{d}$. This yields the shortest distance between the two parallel lines.

Based on the advisor's response, we need to follow these steps to find the distance between the two parallel lines:

1. Identify a point on each line. For the first line, when $t = 0$, the point is $\mathbf{a}_1 = (3, -10, 1)^T$. For the second line, when $u = 0$, the point is $\mathbf{a}_2 = (-5, -3, 6)^T$.
2. Calculate
$$\vec{PQ} = \mathbf{a}_2 - \mathbf{a}_1 = \begin{pmatrix} -8 \\ 7 \\ 5 \end{pmatrix}.$$
3. Use the direction vector $\mathbf{d} = (2, -9, -2)^T$ to find $\vec{PQ} \times \mathbf{d}$.
4. Compute the magnitude of the cross product $\|\vec{PQ} \times \mathbf{d}\|$.
5. Divide the magnitude of the cross product by $\|\mathbf{d}\|$.

Let's perform these calculations step by step.

[... Omitted for brevity ...]

Thus, the distance between the two lines is

$$\boxed{7}.$$

Transform

The SLM initially attempts to use an inappropriate cross-product formula for the distance between parallel lines and incorrectly interprets the zero cross product of the direction vectors. The advisor redirects the SLM to the correct point-to-line distance formula using a connecting vector and a single direction vector. The SLM then carries out the numerical computation independently and obtains the final answer 7.

MATH500, Problem 287 Advisor budget: 512 tokens

Original problem

One gear turns $33\frac{1}{3}$ times in a minute. Another gear turns 45 times in a minute. Initially, a mark on each gear is pointing due north. After how many seconds will the two gears next have both their marks pointing due north?

To determine after how many seconds the two gears will next have both their marks pointing due north, we need to find the least common multiple (LCM) of the number of turns each gear makes in a minute and then convert that time into seconds.
First, let's convert the number of turns per minute for each gear into a more manageable form. The first gear turns $33\frac{1}{3}$ times in a minute, which can be written as $\frac{100}{3}$ turns per minute. The second gear turns 45 times in a minute.

The LCM of $\frac{100}{3}$ and 45 will give us the number of turns per minute after which both gears will be at the same position relative to their starting points. To find the LCM, we need to find the LCM of the numerators when the fractions are expressed with a common denominator.
The first gear's turns per minute is $\frac{100}{3}$, and the second gear's turns per minute is $\frac{135}{3}$, since $45 = \frac{135}{3}$. The LCM of 100 and 135 will be the numerator of the LCM of the fractions.
The prime factorizations are

$$100 = 2^2 \times 5^2$$

and

$$135 = 3^3 \times 5.$$

The LCM of 100 and 135 is

$$2^2 \times 3^3 \times 5^2 = 1800.$$

So, the LCM of $\frac{100}{3}$ and $\frac{135}{3}$ is $\frac{1800}{3} = 600$ turns per minute. This means that after 600 turns, both gears will be at the same position relative to their starting points.
Since there are 60 seconds in a minute, the time in seconds is

$$600 \text{ turns} \times \frac{1 \text{ minute}}{60 \text{ seconds}} = 10 \text{ minutes} = 10 \times 60 = 600 \text{ seconds}.$$

However, we need to confirm that this is the smallest such time. To do this, we can use the advisor to verify the calculation and ensure we have the correct LCM.

QMaker-generated query

Request. Two gears turn $33\frac{1}{3}$ times and 45 times in a minute, respectively. Initially, a mark on each gear is pointing due north. After how many seconds will the two gears next have both their marks pointing due north?

Budget instruction. Answer in 5–7 lines.

Advisor response

First, convert the rotation rates to revolutions per second: Gear A turns $33\frac{1}{3} = \frac{100}{3}$ times per minute, which is $\frac{100}{180} = \frac{5}{9}$ revolutions per second. Gear B turns 45 times per minute, which is $\frac{45}{60} = \frac{3}{4}$ revolutions per second.
For both marks to point north again, the elapsed time must be a common multiple of the time each gear takes to complete one full revolution. The period for Gear A is $\frac{9}{5}$ seconds, and for Gear B is $\frac{4}{3}$ seconds.
We need the least common multiple of these two fractions:

$$\text{LCM}\left(\frac{9}{5}, \frac{4}{3}\right) = \frac{\text{LCM}(9,4)}{\text{GCD}(5,3)}.$$

The LCM of the numerators 9 and 4 is 36, and the GCD of the denominators 5 and 3 is 1. Thus, the time required is

$$\frac{36}{1} = 36$$

seconds. This is the first time both gears simultaneously return their marks to the north position.

The advisor has confirmed that the least common multiple of the periods of the two gears is 36 seconds. Therefore, the two gears will next have both their marks pointing due north after

$$\boxed{36}$$

seconds.

Full-task

The QMaker does not isolate a missing subproblem or request a specific piece of guidance. It simply restates the original benchmark problem and asks for its answer. The advisor then solves the task end to end. The SLM performs no additional derivation after the advisor call and merely repeats the advisor's conclusion. The efficiency gain in this case solely comes from the budget instruction.

LiveCodeBench v6, Problem 236 Advisor budget: 1024 tokens

Original problem

You are an expert Python programmer. You will be given a question (problem specification) and will generate a correct Python program that matches the specification and passes all tests.
Below is the question:
There are N cities in a certain country. You will travel from your office in city 1 to a destination in city N, via zero or more cities. Two types of transportation are available: company car and train. The time required to travel from city i to city j is as follows:
- $D_(i,j) \times A$ minutes by company car, and - $D_-(i,j) \times B + C$ minutes by train.
You can switch from company car to train, but not vice versa. You can do so without spending time, but only in a city. What is the minimum time in minutes to travel from city 1 to city N?

[...Constraints, Input, Output, Examples, ...]

Student turn 1

```
import sys
from typing import List

def read_input() -> List[List[int]]:
    n, a, b, c = map(int, sys.stdin.readline().split())
    d = []
    for _ in range(n):
        d.append(list(map(int, sys.stdin.readline().split())))
    return n, a, b, c, d

def min_time_to_travel(n: int, a: int, b: int, c: int, d: List[List[
int]]) -> int:
    # Initialize the distance matrix
    dist = [[float('inf')] * n for _ in range(n)]
    for i in range(n):
        dist[i][i] = 0

    # Fill the distance matrix using the given distances
    for i in range(n):
        for j in range(n):
            if d[i][j] > 0:
                dist[i][j] = min(d[i][j] * a, d[i][j] * b + c)

    # Floyd-Warshall algorithm to find the shortest path
    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist[0][n-1]

if __name__ == "__main__":
    n, a, b, c, d = read_input()
    result = min_time_to_travel(n, a, b, c, d)
```

```
    print(result)
```

QMaker-generated query

Request. Given: - N cities in a country. - You start in city 1 and want to travel to city N. - Two types of transportation are available: company car and train. - The time required to travel from city i to city j is: - $D_{i,j} \times A$ minutes by company car, and - $D_{i,j} \times B + C$ minutes by train. - You can switch from company car to train, but not vice versa. - You can switch without spending extra time, but only in a city.
How would you compute the minimum time in minutes to travel from city 1 to city N?

Budget instruction. 20 lines

Advisor response

1. Define $D_{i,j}$ as the distance between city i and city j.
2. Let $T_{car}(i,j) = D_{i,j} \times A$ and $T_{train}(i,j) = D_{i,j} \times B + C$.
3. Since switching is allowed only once (car to train), split the path into two segments.
4. Segment 1: Travel from city 1 to an intermediate city k using only the company car.
5. Segment 2: Travel from city k to city N using only the train.
6. The total time for a specific switch point k is $Time(k) = \min_{path_1} \sum T_{car} + \min_{path_2} \sum T_{train}$.
7. Compute the shortest path from city 1 to all cities k using only car edges (Dijkstra).

[... Omitted for brevity ...]

19. The constraint "switch only once" is satisfied by iterating over every possible switch city k.
20. Return the computed minimum time as the solution.

Student turn 2

```
import sys
from typing import List

def read_input() -> List[List[int]]:
    n, a, b, c = map(int, sys.stdin.readline().split())
    d = []
    for _ in range(n):
        d.append(list(map(int, sys.stdin.readline().split())))
    return n, a, b, c, d

def min_time_to_travel(n: int, a: int, b: int, c: int, d: List[List[
int]]) -> int:
    # Initialize the distance matrix
    dist_car = [[float('inf')] * n for _ in range(n)]
    dist_train = [[float('inf')] * n for _ in range(n)]

    # Fill the distance matrix using the given distances
    for i in range(n):
        for j in range(n):
            if d[i][j] > 0:
                dist_car[i][j] = d[i][j] * a
                dist_train[i][j] = d[i][j] * b + c

    # Compute the shortest path from city 1 to all cities using only
car edges
    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist_car[i][j] = min(dist_car[i][j], dist_car[i][k] +
 dist_car[k][j])
```

```
    # Compute the shortest path from all cities to city N using only
train edges
    for k in range(n):
        for i in range(n):
            for j in range(n):
                dist_train[i][j] = min(dist_train[i][j], dist_train[i
][k] + dist_train[k][j])

    # Find the minimum time by considering all possible switch points
    min_time = float('inf')
    for k in range(n):
        if k != 0 and k != n-1:
            time = dist_car[0][k] + dist_train[k][n-1]
            min_time = min(min_time, time)
        elif k == 0:
            time = dist_train[0][n-1]
            min_time = min(min_time, time)
        elif k == n-1:
            time = dist_car[0][n-1]
            min_time = min(min_time, time)

    return min_time

if __name__ == "__main__":
    n, a, b, c, d = read_input()
    result = min_time_to_travel(n, a, b, c, d)
    print(result)
```

Full-task

The advisor provides the solution strategy entirely in natural-language steps, describing how to separate the car and train shortest-path computations around a single switch city. However, the SLM fails to faithfully integrate this guidance into the final code. The student utilizes Floyd–Warshall computation, instead of suggested Dijstra-based approach. This example illustrates one example of integration failure by the SLM.