BEACONAGENT: LEARNING CAPABILITY-AWARE HELP-SEEKING POLICIES FOR SMALL LANGUAGE MODEL AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Small language model (SLM) agents offer a resource-efficient alternative to large language model (LLM) agents, but their limited capabilities make them unreliable on challenging tasks. Recent work enables SLM agents to leverage stronger LLMs through self-reflection, confidence estimation, and learned routing. However, existing approaches mainly focus on assistance routing, with limited attention to the design of a fine-grained help-seeking policy. First, the critical step where assistance is most valuable may fail to be identified. Second, assistance decisions may not align with the SLM agent's capability boundary, causing unnecessary or insufficient help. Third, existing approaches often rely on delegation, increasing LLM involvement and reducing SLM autonomy. To address these challenges, we propose **BeaconAgent**, which enables the SLM to execute the full agent trajectory while selectively requesting brief, targeted LLM guidance provided only as an observation. BeaconAgent learns this help-seeking behavior through two training stages. First, a teacher LLM identifies the **first critical error** causing trajectory failure, replaces it with an explicit `ASK_HELP` step, and completes the remaining trajectory with guidance. The resulting successful trajectories are used for supervised fine-tuning, teaching the SLM agent **where and how to ask**. Second, capability-aware reinforcement learning aligns help seeking with the SLM agent's capability boundary, promoting assistance when needed while reducing unnecessary help otherwise, thereby teaching the SLM agent **when to ask**. We evaluate BeaconAgent on math reasoning, retrieval-augmented QA, and unseen coding tasks. Results show that BeaconAgent improves task performance while reducing unnecessary LLM usage, achieving a better performance–cost trade-off.

1 INTRODUCTION

Large language model (LLM)-based agents have demonstrated strong capabilities in reasoning, tool use, and complex multi-step tasks (Yao et al., 2023; Schick et al., 2023), but deploying them throughout the full execution trajectory incurs substantial inference costs. Small language model (SLM) agents are more efficient but less reliable on challenging tasks. To combine the efficiency of SLMs with the capability of LLMs, prior studies have explored selective model utilization strategies, where lightweight models dynamically leverage stronger models according to task difficulty, expected utility, or estimated failure risk (Chen et al., 2024; Ong et al., 2025; Ding et al., 2024), improving the performance–cost trade-off. More recently, this paradigm has been extended to agent settings, where SLM agents seek assistance from stronger LLMs during execution. Existing agent-level approaches typically use self-reflection, confidence estimation, or learned routing to determine when external assistance is needed (Sun et al., 2024; Lee et al., 2026; Hemadri et al., 2026).

However, the above approaches still face several limitations. First, the signals used to trigger help seeking may not accurately capture the trajectory state where assistance is most valuable (Lee et al., 2026). A locally uncertain step may be recoverable, whereas another seemingly reasonable action may become the critical decision leading to failure. Therefore, an effective help-seeking agent should learn to perceive whether an upcoming action is likely to cause failure and request assistance before the error propagates. Second, learning help-seeking behavior from failed cases or relying on

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.

external routing criteria may not fully align assistance decisions with the SLM agent's capability boundary (Hemadri et al., 2026). The agent should request help when it cannot solve a task independently while avoiding unnecessary assistance within its capability. Third, existing approaches mainly rely on switching or delegation mechanisms (Sun et al., 2024), where stronger LLM agents directly generate or correct current reasoning steps. Such delegation increases LLM involvement and weakens SLM autonomy. A more efficient paradigm is to let the SLM agent execute the full agent trajectory while selectively requesting brief, targeted LLM guidance when necessary.

To address these challenges, we propose **BeaconAgent**, a two-stage framework that enables SLM agents to learn selective and minimal assistance seeking. In the first stage, we exploit failed trajectories generated by the SLM agent as supervision signals. Specifically, a teacher LLM agent identifies the **first critical error** that causes the trajectory to deviate toward failure. We replace this failure-causing step with an explicit `ASK_HELP` action, obtain brief teacher-generated guidance as an observation, and continue execution from the assisted state. Successful repaired trajectories are collected for supervised fine-tuning, teaching the SLM agent **where and how** to ask for help while preserving its role as the main executor. In the second stage, we introduce capability-aware reinforcement learning to align help-seeking behavior with the SLM agent's actual capability. Tasks that the SLM agent cannot solve independently encourage help seeking, whereas tasks within its capability reward autonomous execution. Through this process, BeaconAgent learns not only **when** assistance is useful but also **when** assistance is unnecessary.

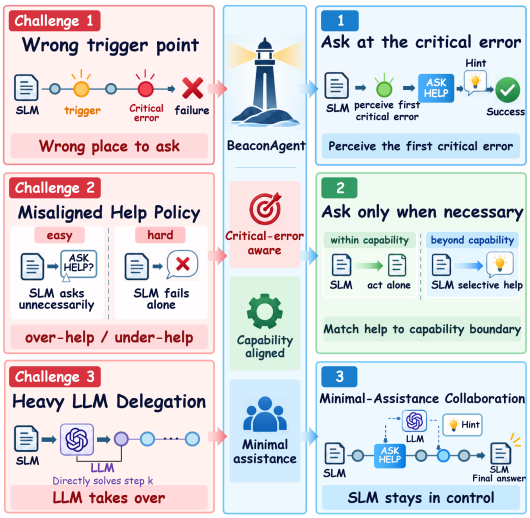


Figure 1: Overview of BeaconAgent, which addresses three key limitations of existing SLM–LLM collaboration through critical-error awareness, capability-aligned help seeking, and minimal assistance.

We evaluate BeaconAgent on math reasoning and QA tasks, and further investigate generalization on unseen coding tasks. Beyond task performance, we analyze help-seeking behavior, LLM usage, and the performance–cost trade-off. At its core, BeaconAgent distills prior failure experience into an SLM-native help-seeking policy, rather than detecting errors at runtime and handing execution over to an LLM, enabling stronger agent capabilities with substantially lower LLM involvement. Experimental results demonstrate that BeaconAgent enables SLM agents to autonomously perceive potential critical errors, request targeted assistance at these steps, and maintain autonomous execution as much as possible, achieving more efficient and reliable SLM–LLM collaboration.

Our contributions are summarized as follows:

- **Minimality:** We propose an SLM–LLM collaboration paradigm in which the SLM remains the primary executor and receives only brief, targeted LLM guidance instead of delegating execution.
- **Targeting:** We introduce a failure-driven supervision strategy that identifies the first critical error causing trajectory deviation and converts it into a proactive `ASK_HELP` action, teaching the SLM where and how to seek assistance.
- **Selectivity:** We develop a capability-aware reinforcement learning approach that aligns help seeking with the SLM agent's capability boundary, teaching the SLM when assistance is necessary and when autonomous execution is sufficient.

2 METHOD

We propose **BeaconAgent**, a two-stage framework that trains a small language model (SLM) agent to seek brief, targeted help from a stronger teacher LLM while remaining the primary task executor.

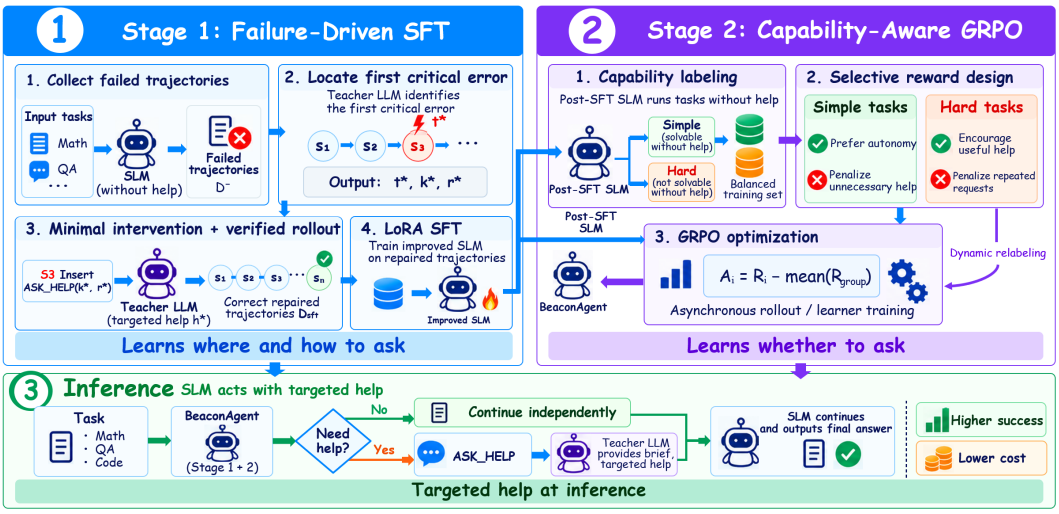


Figure 2: **Stage 1:** Failure-driven SFT enables SLM to learn where and how to ask for help. **Stage 2:** Capability-aware GRPO enables SLM to learn whether to ask for help. **During inference**, the SLM requests brief teacher guidance when needed and continues execution independently.

Let π_θ denote the SLM policy and $\mathcal{T}$ the teacher. At any intermediate state s_t, the SLM can either continue acting independently or issue

$$a_t^{\text{help}} = \texttt{ASK_HELP}(k_t, r_t), \qquad k_t \in \mathcal{K} = \{\text{PLAN}, \text{VERIFY}, \text{REPAIR}\}, \tag{1}$$

where k_t requests a next-steps PLAN, a VERIFY check of intermediate results or evidence, or a local-error REPAIR (See Appendix A.8 for details). The SLM-written request r_t identifies a potentially flawed reasoning step, calculation, evidence use, or strategy and the limited guidance needed. Given the task, trajectory prefix, k_t, and r_t, the teacher returns a short hint h_t, after which the SLM continues independently. Training comprises **failure-driven SFT** to learn *where and how to ask* from successful trajectory repairs, followed by **capability-aware GRPO** to learn *whether to ask*, favoring autonomy within the SLM's capability and brief, useful help beyond it.

2.1 Failure-Driven Supervised Fine-Tuning

Collecting failed trajectories. We first run the SLM agent without access to ASK_HELP on the mathematics and retrieval-augmented QA training sets. For a task–answer pair (x, y), a trajectory is

$$\tau^- = (s_0, a_0, o_0, \ldots, s_T, a_T, o_T), \tag{2}$$

where a_t and o_t denote an agent action and its resulting observation. An answer judge $C(\hat{y}, y) \in \{0, 1\}$ evaluates the final answer. We retain only failed trajectories,

$$\mathcal{D}^- = \{(x, y, \tau^-) : C(\text{Ans}(\tau^-), y) = 0\}. \tag{3}$$

Locating the first critical error. The teacher examines each complete failed trajectory and returns

$$\mathcal{J}_\mathcal{T}(x, y, \tau^-) = (t^*, k^*, r^*), \qquad \tau^- \in \mathcal{D}^-, \tag{4}$$

where t^* denotes the step containing the **first critical error** (See Appendix A.7 for details): the assistant step with a genuine reasoning, evidence-use, or strategy error that drives subsequent failure yet remains recoverable by one local intervention:

$$t^* = \min\{t : \text{Critical}(a_t, \tau^-) = 1 \ \wedge \ \text{LocallyRecoverable}(s_t) = 1\}. \tag{5}$$

The teacher first identifies the root cause of failure, then rewrites it as r^*, a first-person help request from the SLM's perspective that describes the potential error and asks for limited guidance. It also selects the corresponding assistance type $k^* \in \mathcal{K}$.

Minimal intervention and verified rollout. We preserve the SLM prefix $\tau^-_{<t^*}$ and replace the critical action with $\texttt{ASK_HELP}(k^*, r^*)$. The teacher provides

$$h^* \sim \mathcal{T}_{\text{help}}(x, \tau^-_{<t^*}, k^*, r^*), \tag{6}$$

where h^* is a short, local, actionable hint containing neither the final answer nor a full solution (Appendix A.6 for specific definition). The repaired prefix is

$$\widetilde{\tau}_{\le t^*} = \tau^-_{<t^*} \oplus [\texttt{ASK_HELP}(k^*, r^*), h^*]. \tag{7}$$

The teacher agent then completes the remaining interaction:

$$\widetilde{\tau} = \widetilde{\tau}_{\le t^*} \oplus \rho, \qquad \rho \sim \mathcal{T}_{\text{rollout}}(\cdot \mid x, \widetilde{\tau}_{\le t^*}). \tag{8}$$

We retain only well-formed trajectories with correct final answers:

$$\mathcal{D}_{\text{SFT}} = \{\widetilde{\tau} : C(\text{Ans}(\widetilde{\tau}), y) = 1 \ \wedge \ \text{Valid}(\widetilde{\tau}) = 1\}. \tag{9}$$

Supervised learning. We LoRA-fine-tune (Hu et al., 2022) the same SLM on retained multi-turn trajectories. The mask $\mathcal{M}(\widetilde{\tau})$ selects only assistant-generated tokens, excluding system, user, and tool-observation tokens, for next-token loss:

$$\mathcal{L}_{\text{SFT}}(\theta) = -\sum_{\widetilde{\tau} \in \mathcal{D}_{\text{SFT}}} \sum_{j} \mathcal{M}_j(\widetilde{\tau}) \log \pi_\theta(w_j \mid w_{<j}). \tag{10}$$

Targets include pre-help reasoning, the request type and text, and post-help continuation, teaching where to ask, how to phrase requests, and how to use hints.

2.2 Capability-Aware Group Relative Policy Optimization

Capability-Aware Task Construction. Because every SFT demonstration includes help, SFT may encourage excessive requests. To estimate the capability boundary, we independently sample four trajectories at temperature 1.0, $\tau_{x,j}^{\text{nohelp}} \sim \pi_{\text{SFT}}(\cdot \mid x, \text{help disabled})$, $j = 1, \ldots, 4$, for each task. Tasks with at least three incorrect final answers are labeled H; all others are labeled S:

$$d_0(x) = \text{H if } \sum_{j=1}^{4} \Big[1 - C(\text{Ans}(\tau_{x,j}^{\text{nohelp}}), y)\Big] \ge 3, \quad \text{S otherwise.} \tag{11}$$

Here S (simple) and H (hard) reflect SLM capability, not intrinsic difficulty. Through this capability assessment and balanced selection across domains and capability labels, we collect 6,000 qualifying tasks for GRPO, equally distributed among (math, S), (math, H), (QA, S), and (QA, H) and interleaved in training. For the 3,000 hard tasks, we rerun the post-SFT SLM and the teacher LLM through *Locating the first critical error* and *Minimal intervention and verified rollout* (Section 2.1), obtaining one verified help-containing reference trajectory per task. We record its inserted `ASK_HELP` step as t_x^*, the reference critical-error position used for RL alignment below. As RL improves the SLM, previously unsolvable tasks may become solvable without help, motivating dynamic relabeling to keep capability labels valid. Let $\mathcal{G}_x$ be the retained rollout group collected by the behavior policy, $G_x = |\mathcal{G}_x|$, and z_i, n_i its trajectories' final correctness and help-request counts. The unassisted success rate is

$$\widehat{p}_0(x) = \frac{1}{G_x} \sum_{i \in \mathcal{G}_x} \mathbb{I}[z_i = 1 \wedge n_i = 0]. \tag{12}$$

Let $G_{\min}$ and G_{target} denote the minimum eligible and target group sizes, and let $e_x = \mathbb{I}[G_{\min} \le G_x \le G_{\text{target}}]$. The effective label shared by the retained group is

$$d(x) = \text{S if } d_0(x) = \text{S or } (e_x = 1 \wedge \widehat{p}_0(x) > \tau_{\text{rel}}), \quad \text{H otherwise.} \tag{13}$$

where τ_{rel} is a strict threshold. The denominator includes all retained trajectories, not only those without help. This group-local, one-way override leaves the offline label unchanged and permits eligible partial groups; total correctness alone does not trigger relabeling.

Capability-Aware Help-Seeking Reward. For rollout i, let $u_i \in \{\text{math}, \text{QA}\}$ be the task domain, $z_i \in \{0, 1\}$ its correctness, n_i its number of requests, and $q_i = \mathbb{I}[n_i > 0]$. Let λ_{rep} denote the penalty for every request after the first. The base reward implemented by our selective reward profile is

$$R_i^{\text{base}} = \begin{cases} r_{0,z_i}^{\text{S},u_i}, & d(x_i) = \text{S},\ q_i = 0, \\ r_{1,z_i}^{\text{S},u_i} - \lambda_{\text{rep}}(n_i - 1), & d(x_i) = \text{S},\ q_i = 1, \\ r_{0,z_i}^{\text{H}}, & d(x_i) = \text{H},\ q_i = 0, \\ b_{z_i}^{\text{H}} - \lambda_{\text{rep}}(n_i - 1), & d(x_i) = \text{H},\ q_i = 1, \end{cases} \tag{14}$$

Request types do not affect rewards. Simple-task rewards are domain-specific, including autonomous success, whereas hard-task rewards are shared across domains. The constants satisfy:

$$r_{0,1}^{\text{S},u} > r_{1,1}^{\text{S},u}, \qquad \text{prefer autonomy on simple tasks,} \tag{15}$$

$$b_1^{\text{H}} > r_{0,1}^{\text{H}}, \qquad \text{preserve useful help on hard tasks,} \tag{16}$$

To discourage help-seeking positions from drifting during RL, we compare the first help request in each hard-task rollout with its reference position t_x^*. For rollout τ_i, define

$$t_i^{\text{ask}} = \min\{t : a_{i,t} = \texttt{ASK_HELP}(k, r)\}, \qquad \min \emptyset = +\infty. \tag{17}$$

Using the same assistant-step indexing for reference and rollout trajectories, we assign a small positive reward $R_{\text{ali},1} > 0$ for an exact match and a small negative reward $R_{\text{ali},2} < 0$ otherwise:

$$R_i^{\text{align}} = \begin{cases} R_{\text{ali},1}, & d(x_i) = \text{H},\ n_i > 0,\ t_i^{\text{ask}} = t_{x_i}^*, \\ R_{\text{ali},2}, & d(x_i) = \text{H},\ n_i > 0,\ t_i^{\text{ask}} \neq t_{x_i}^*, \\ 0, & \text{otherwise.} \end{cases} \tag{18}$$

Rollouts without help and tasks labeled as simple receive no alignment reward or penalty. We add this term before outcome clipping, $R_i^{\text{aug}} = R_i^{\text{base}} + R_i^{\text{align}}$. Repetition penalties are applied before clipping. Correctness remains primary: the implementation enforces a success floor $R_{\min}^+$ and a failure ceiling $R_{\max}^-$ with $R_{\min}^+ > R_{\max}^-$,

$$\overline{R}_i = \max(R_i^{\text{aug}}, R_{\min}^+) \text{ if } z_i = 1, \quad \min(R_i^{\text{aug}}, R_{\max}^-) \text{ if } z_i = 0. \tag{19}$$

Reward preference constraints and the main optimization configurations are in the appendix A.2.

Group-Relative Policy Optimization. We sample eight RL rollouts per task at temperature 1.0 ($G_{\text{target}} = 8$). For each task x, policy μ samples multi-turn trajectories interleaving SLM generations, tool observations, and teacher help; $\mathcal{G}_x$ denotes the retained group. After answer judging and Eqs. equation 13–equation 19, we center rewards $\overline{R}_i$ within the group without standard-deviation normalization:

$$A_i = \overline{R}_i - \frac{1}{G_x} \sum_{j \in \mathcal{G}_x} \overline{R}_j. \tag{20}$$

Let $m_{i,t} \in \{0, 1\}$ mask policy tokens: it is one only for trainable SLM-generated tokens and zero for tool and teacher-help observations and the untrained forced-final response. Let $L_i^\pi = \sum_{t=1}^{|\tau_i|} m_{i,t}$. For each position with $m_{i,t} = 1$, let $\rho_{i,t}(\theta) = \pi_\theta(a_{i,t} \mid s_{i,t}) / \mu(a_{i,t} \mid s_{i,t})$ be the token-level importance ratio. We optimize the sequence-mean, policy-token-mean dual-clipped objective. Let $\ell_{i,t}^{\text{clip}}(\theta) = \min\{\rho_{i,t}(\theta) A_i, \text{clip}(\rho_{i,t}(\theta), 1-\epsilon, 1+\epsilon) A_i\}$ denote the standard clipped surrogate. With a dual-clip coefficient $c_{\text{dual}} > 1$, the token-level objective is

$$\ell_{i,t}(\theta) = \ell_{i,t}^{\text{clip}}(\theta) + \mathbb{I}[A_i < 0] \max\{c_{\text{dual}} A_i - \ell_{i,t}^{\text{clip}}(\theta), 0\}. \tag{21}$$

Let $\mathcal{B}$ denote the retained trajectories with $L_i^\pi > 0$ in a learner minibatch. We average over these trajectories rather than equally weighting task groups:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}_{\mathcal{B}} \left[\frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \frac{1}{L_i^\pi} \sum_{t=1}^{|\tau_i|} m_{i,t} \ell_{i,t}(\theta) \right]. \tag{22}$$

In our fully asynchronous rollout–learner pipeline, workers execute multi-turn agents and external tools while the learner updates the LoRA policy; weights synchronize periodically with bounded staleness. The importance ratio corrects for each trajectory's behavior policy. No extra KL or entropy term is added. Sampling, optimization, clipping, asynchrony, and LoRA hyperparameters appear in the appendix A.2.

3 Experiments

3.1 Experimental Setup

Training data. Our failure-driven SFT set contains 3,021 verified, successfully repaired agent trajectories: 1,419 from the MATH training data and 1,602 from HotpotQA training data. Each demonstration retains the trajectory prefix before the first critical error, inserts a targeted `ASK_HELP` interaction, and provides a successful continuation, as specified in Section 2.1. For capability-aware GRPO, we use 6,000 tasks from MATH and HotpotQA training data, with 3,000 tasks per domain. We assign capability labels using unassisted rollouts of the post-SFT policy. The resulting four domain–capability strata each contain 1,500 tasks (Section 2.2).

Models and agent implementation. We use Qwen3-4B as the local SLM and Qwen3.6-27B-FP8 as the teacher and assisting LLM, with thinking mode disabled for both models by default. The agent is implemented with CodeAct (Wang et al., 2024), which executes code-based actions and interacts with task-relevant tools. We set the maximum trajectory length to seven agent steps. BeaconAgent is trained with LoRA for two SFT epochs, followed by one epoch of capability-aware GRPO.

Evaluation benchmarks. We evaluate on eleven benchmarks spanning three domains: (i) mathematical reasoning on MATH500 (Hendrycks et al., 2021), AIME, Gaokao (Zhong et al., 2024),

OlyMath (Sun et al., 2026), and Omni-MATH (Gao et al., 2025); (ii) retrieval-augmented QA on HotpotQA (Yang et al., 2018), 2WikiMultiHopQA (2Wiki) (Ho et al., 2020), Bamboogle (Press et al., 2023), MuSiQue (Trivedi et al., 2022), and Natural Questions (NQ) (Kwiatkowski et al., 2019); and (iii) code generation on HumanEval (Chen et al., 2021). More details can be found in appendix A.3.

Baselines. We compare with three SLM–LLM collaboration methods. **AdaSwitch** (Sun et al., 2024) learns step-level self-reflection and adaptively switches to a stronger agent to correct problematic local steps. **R2vAgent** (Hemadri et al., 2026) combines a trained local policy and an action verifier with a calibrated router that estimates residual failure risk and triggers teacher intervention. **STEER** (Lee et al., 2026) uses model-internal confidence for stepwise routing between small and large models, without training an external router. Following the original papers, we faithfully reimplement the three baselines under our experimental setting. For methods requiring training, we use the same underlying training tasks as BeaconAgent while following each method's original training procedure to ensure a fair comparison. We additionally report Qwen3-4B with and without access to help, BeaconAgent after only SFT with help enabled, and the full BeaconAgent policy with help disabled. A standalone Qwen3.6-27B-FP8 agent provides a large-model reference.

Table 1: Pass@1 accuracy (%). All SLM-based methods use Qwen3-4B. Bold marks the best SLM-based result, including ties; Qwen3.6-27B-FP8 is a standalone large-model reference. All methods have help enabled unless marked otherwise.

Mathematical reasoning and code generation

Method	MATH	AIME	Gaokao	OlyMath	Omni-MATH	HumanEval
Qwen3-4B (no help)	66.0	16.7	46.5	11.0	28.3	78.7
Qwen3-4B	72.2	21.1	64.2	12.5	31.0	80.5
AdaSwitch	73.2	32.2	65.7	15.5	34.8	76.2
R2vAgent	80.4	28.9	74.0	16.0	36.8	82.3
STEER	82.0	**34.4**	70.9	**18.0**	37.4	74.4
BeaconAgent (SFT only)	80.6	33.3	65.3	11.0	37.8	78.7
BeaconAgent (no help)	83.0	26.6	73.0	13.0	36.6	78.0
BeaconAgent	**86.4**	33.3	**76.1**	**18.0**	**41.3**	**85.4**
Qwen3.6-27B-FP8	94.4	74.4	89.0	51.0	50.0	95.7

Retrieval-augmented QA and overall performance

Method	HotpotQA	2Wiki	Bamboogle	MuSiQue	NQ	Avg.Acc
Qwen3-4B (no help)	51.2	43.6	49.6	17.2	57.2	42.36
Qwen3-4B	57.8	51.4	63.2	24.6	62.0	49.14
AdaSwitch	57.0	45.8	55.2	20.0	50.2	47.80
R2vAgent	61.6	56.8	55.2	29.1	**65.4**	53.32
STEER	51.6	46.4	48.0	19.0	52.4	48.59
BeaconAgent (SFT only)	61.8	52.4	60.8	26.4	60.6	51.70
BeaconAgent (no help)	63.4	53.4	63.2	24.8	62.0	52.45
BeaconAgent	**67.6**	**57.0**	**71.2**	**31.4**	**65.4**	**57.55**
Qwen3.6-27B-FP8	70.0	54.2	72.0	42.4	64.8	68.90

3.2 Main Results

Overall performance. Table 1 shows that BeaconAgent achieves the highest average accuracy among the SLM-based configurations, reaching 57.55%. It exceeds AdaSwitch, R2vAgent, and STEER by 9.75, 4.24, and 8.96 percentage points, respectively. Relative to the base Qwen3-4B agent, BeaconAgent improves average accuracy by 15.19 points and by 8.42 points when the base model also has access to help. BeaconAgent attains the best or joint-best SLM-based accuracy on ten of the eleven benchmarks. The gains span math reasoning and retrieval-augmented QA, with particularly strong results on MATH (86.4%), HotpotQA (67.6%), and Bamboogle (71.2%). On the unseen coding domain, BeaconAgent reaches 85.4% on HumanEval, compared with 82.3% for the strongest collaboration baseline, R2vAgent.

Effects of training and assistance. The SFT-only variant reaches 51.70% average accuracy, while the full BeaconAgent improves this to 57.55%, a gain of 5.85 points after capability-aware GRPO. With help disabled, the trained BeaconAgent still achieves 52.45%, compared with 42.36% for the unassisted base model. Enabling help for the full policy adds another 5.10 points. These results indicate that the final system combines a stronger autonomous policy with useful assistance at inference time.

Assistance efficiency. As shown in Table 7, BeaconAgent consumes only 49.39 LLM output tokens per assisted trajectory, compared with 403.00 for AdaSwitch, 3,628.39 for R2vAgent, and 663.78 for STEER. This corresponds to reductions of 87.7%, 98.6%, and 92.6%, respectively. BeaconAgent also has the smallest reported aggregate LLM output-token total. Crucially, these reductions accompany the highest average accuracy among the four methods. Figure 3 illustrates the performance–cost trade-off between task accuracy and LLM total output-token consumption. These results demonstrate our key contribution: by restricting the teacher LLM to brief, targeted guidance, BeaconAgent substantially reduces inference cost while still delivering significant performance gains.

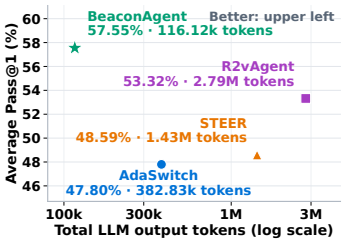


Figure 3: Average Pass@1 vs total LLM output tokens.

3.3 Analysis I: Where to Ask—Critical-Step Alignment

Table 2: Critical-step alignment. N is the number of selected pairs for the alignment analysis. Acc. denotes the overall benchmark accuracy computed on the full benchmark. Acc. and Hit@0 are percentages; Distance is measured in agent steps.

	Random-SFT			Critical-Error SFT			BeaconAgent		
Benchmark (N)	Acc.↑	Hit@0↑	Dist.↓	Acc.↑	Hit@0↑	Dist.↓	Acc.↑	Hit@0↑	Dist.↓
MATH500 (50)	75.7	50.0	0.56	80.6	74.0	0.40	**86.4**	**78.0**	**0.38**
HotpotQA (100)	61.2	41.0	0.94	61.8	56.0	0.73	**67.6**	**57.0**	**0.69**
2Wiki (50)	52.2	34.0	1.04	52.4	**52.0**	0.78	**57.0**	50.0	**0.76**
Gaokao (50)	64.9	62.0	0.42	65.3	76.0	**0.34**	**76.1**	**78.0**	0.38
HumanEval (20)	64.6	5.0	1.00	78.7	**20.0**	0.90	**85.4**	15.0	**0.85**

Controlled comparison. We compare **Random-SFT** and **Critical-Error SFT**, both trained on the same 3,021 examples using Qwen3-4B. Random-SFT places `ASK_HELP` at a random eligible step, whereas Critical-Error SFT places it at the first critical error. The teacher, hint-generation protocol, training size, epochs, LoRA configuration, and learning rate are held fixed.

Critical-step evaluation. For each model, we pair an unassisted rollout with a help-enabled rollout of the same task, retaining cases where the former fails and the latter requests help. The teacher model identifies the first critical failure step t_i^* in the unassisted trajectory; we compare this index with the help-request step t_i^{ask} in the assisted trajectory. Over the N selected cases, we report **Hit@0**, the fraction for which the help-request step exactly matches the first critical failure step, and **Mean Distance**, the average absolute difference between these two step indices, measured in agent steps.

Key findings. Critical-Error SFT improves Hit@0 over Random-SFT on all five benchmarks by 14–24 percentage points and reduces mean distance on every benchmark (Table 2). Accuracy also improves throughout, including gains of 4.9 points on MATH500 and 14.1 points on HumanEval. This directly demonstrates the contribution of our method: supervising help requests at the first critical error is more effective than doing so at an arbitrary eligible step. BeaconAgent further improves accuracy on all five benchmarks while maintaining broadly stable Hit@0 and mean distance, with modest improvements on several benchmarks. These results suggest that capability-aware GRPO improves task performance while preserving the critical-step alignment learned during SFT.

3.4 Analysis II: Whether to Ask—Capability Alignment

Evaluation protocol. We re-evaluate the model's capabilities and help-seeking behavior on test questions not included in the training set. For each policy separately, we first disable `ASK_HELP`:

questions answered correctly are *within-capability*, and those answered incorrectly are *beyond-capability*. We then enable help on the same questions and report the fraction of trajectories containing at least one help request, both overall and within each subset (Table 3).

Selective help seeking. After GRPO, within-capability ask rates decrease by 22.29–40.92 percentage points across all five benchmarks, while accuracy increases by 4.6–10.8 points. Beyond-capability ask rates increase on three benchmarks; The full policy asks substantially more often on beyond-capability than within-capability questions on every benchmark, with gaps of 23.56–42.71 points. These results validate our contribution: capability-aware GRPO enables BeaconAgent to learn when to seek assistance by distinguishing between within-capability and beyond-capability tasks.

Table 3: Capability-based help seeking (%). Acc.: help-enabled accuracy; Ask: overall ask rate; Within/Beyond: ask rates on each policy's no-help success/failure subsets. Bold marks higher accuracy and lower overall/within-capability ask rates.

Benchmark	BeaconAgent(SFT only)				Full BeaconAgent			
	Acc.	Ask	Within	Beyond	Acc.	Ask	Within	Beyond
MATH500	80.6	72.60	70.54	78.46	**86.4**	**42.20**	**34.94**	77.65
HotpotQA	61.8	67.60	79.78	53.07	**67.6**	**57.00**	**42.60**	82.00
2Wiki	52.4	69.40	67.23	71.37	**57.0**	**62.80**	**44.94**	83.26
Gaokao	65.3	78.40	80.82	75.36	**76.1**	**46.23**	**39.90**	63.46
HumanEval	78.7	64.00	64.75	61.90	**85.4**	**46.30**	**39.80**	69.44

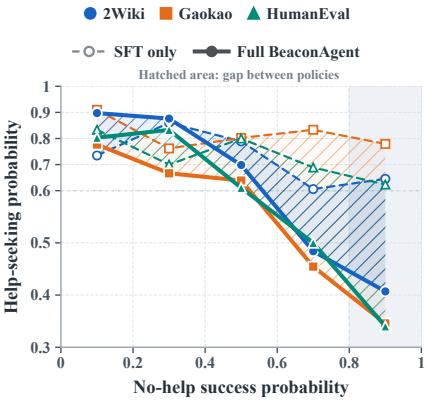


Figure 4: Average help-seeking rate by autonomous success bin.

Repeated-sampling analysis. For each policy on 2Wiki, Gaokao, and HumanEval, we independently sample six no-help and six help-enabled trajectories per test question. We estimate each question's autonomous success rate as the fraction of its six no-help trajectories that yield a correct final answer, and its help-seeking rate as the fraction of its six help-enabled trajectories containing at least one help request. We group questions into five bins by autonomous success rate and average their help-seeking rates within each bin.

Generalization beyond training labels. Figure 4 shows a clear overall decline in the full policy's ask rate as autonomous success increases. From the lowest to the highest capability bin, ask rates fall from 89.74% to 40.70% on 2Wiki, 77.68% to 34.57% on Gaokao, and 80.30% to 34.01% on HumanEval. In the bin with the highest autonomous success rate, SFT still asks at rates of 64.53%, 77.92%, and 62.35%, respectively. Coding is absent from both SFT and GRPO capability labeling, so the HumanEval trend supports transfer beyond training-label memorization.

3.5 Analysis III: How to Ask—Is Minimal Guidance Sufficient?

Controlled comparison. We evaluate the same trained BeaconAgent on the same benchmarks, changing only how the assisting LLM responds to help requests at inference time. With **brief guidance**, the local SLM requests help from the LLM, which returns only a brief, targeted hint as an observation; the SLM then continues executing subsequent steps. With **direct-step execution**, the LLM replaces the SLM at the step where help is requested, executes that step itself, and then returns control to the SLM.

Performance and token cost. Brief guidance achieves 57.55% average accuracy, compared with 56.61% for direct-step execution, a gain of 0.94 percentage points (Table 4). Meanwhile, mean LLM output drops from 356.41 to 49.39 tokens per assisted trajectory, an 86.1% reduction (Figure 5). These results show that brief guidance matches or slightly improves average task performance at substantially lower token cost. The LLM need not generate the difficult step itself: targeted and

Table 4: Comparison between brief guidance and direct-step assistance. LLM tokens denote the mean output tokens generated by the LLM per assisted trajectory.

Benchmark	Brief guidance	Direct step	Δ Accuracy
MATH500	**86.4**	86.0	+0.4
AIME	33.3	**36.6**	-3.3
Gaokao	76.1	**78.4**	-2.3
OlyMath	**18.0**	12.0	+6.0
Omni-MATH	41.3	**44.0**	-2.7
HumanEval	**85.4**	81.7	+3.7
HotpotQA	**67.6**	64.6	+3.0
2Wiki	**57.0**	54.2	+2.8
Bamboogle	**71.2**	66.4	+4.8
MuSiQue	31.4	**35.4**	-4.0
NQ	**65.4**	63.4	+2.0
Average	**57.55**	56.61	+0.94
LLM tokens	**49.39**	356.41	-86.1%

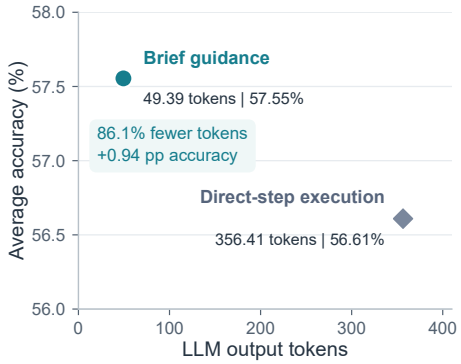


Figure 5: Average accuracy versus mean LLM output tokens per assisted trajectory. Upper left is better.

brief hints can suffice while leaving execution to the SLM. Together with Table 7, these results validate our minimal-assistance collaboration paradigm, demonstrating that SLM agents can achieve competitive or superior performance with substantially fewer LLM tokens. BeaconAgent achieves the best performance while consuming the lowest LLM output tokens among all baselines, highlighting the effectiveness of brief, targeted guidance.

4 Related Work

SLM–LLM collaboration. Recent works study efficient collaboration between SLMs and LLMs by dynamically allocating model capabilities according to task difficulty or expected utility. FrugalGPT (Chen et al., 2024), Hybrid LLM (Ding et al., 2024), and RouteLLM (Ong et al., 2025) investigate cost-aware model routing, aiming to balance inference cost and response quality by selecting appropriate models for different queries. Extending this idea to agent scenarios, AdaSwitch (Sun et al., 2024), STEER (Lee et al., 2026), and R2V-Agent (Hemadri et al., 2026) enable SLM agents to seek assistance from stronger LLMs through mechanisms such as introspection, confidence estimation, or learned triggering. However, these approaches mainly focus on deciding when to invoke external assistance, while the design of a fine-grained help-seeking policy, including where assistance should be provided and how much assistance is sufficient, remains underexplored.

Adaptive behavior learning for LLMs. Recent studies explore adaptive LLM behaviors through feedback and reinforcement learning. RISE (Qu et al., 2024), AGILE (Feng et al., 2024), and ReST-MCTS* (Zhang et al., 2024) improve model behaviors through self-improvement, agent-level reinforcement learning, and process-level optimization, respectively. KALM (Pang et al., 2024) and DeepSeekMath (Shao et al., 2024) further demonstrate the effectiveness of reinforcement learning or model-generated supervision for improving model capabilities. Recent adaptive tool-use studies (Li et al., 2025; Wang et al., 2025) investigate how models can selectively employ external capabilities during problem solving. Different from these works, BeaconAgent focuses on learning a capability-aware help-seeking policy for SLM agents, where failure-driven SFT teaches where and how to ask, and reinforcement learning teaches when assistance is necessary.

5 Conclusion

We presented **BeaconAgent**, a capability-aware help-seeking framework that enables SLM agents to collaborate with stronger LLMs while remaining the primary executor. By combining failure-driven SFT and capability-aware RL, BeaconAgent learns **where, how, and when** to request assistance through critical-error localization, brief targeted guidance, and capability-aligned help seeking. Across eleven benchmarks, including unseen code generation, BeaconAgent improves the performance–cost trade-off by reducing unnecessary LLM involvement while maintaining strong task performance. Our results show that efficient SLM–LLM collaboration can be achieved through minimal assistance rather than execution delegation.

AI USE STATEMENT

In this work, we used generative AI to assist with synthetic data generation and to evaluate the correctness of model outputs on the final test sets. These uses of generative AI are explicitly described in the main text and the appendix. We also used generative AI tools to improve the readability, grammar, and language of the manuscript. The original manuscript and its substantive research content were written by the authors.

We did not use generative AI tools to independently develop the theoretical or conceptual framework, formulate mathematical claims, propose or refine hypotheses, design the overall research methodology or experiments, or interpret the experimental results. All AI-assisted outputs, including generated data, evaluation judgments, and language edits, were reviewed and verified by the authors. In particular, authors carefully reviewed the entire manuscript after AI-assisted editing to ensure that the final text accurately reflected the intended meaning and research findings.

We take full responsibility for the final content of this work, including all text, data, experimental results, claims, analyses, and conclusions produced or presented with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We have made efforts to facilitate the reproducibility of BeaconAgent. The main paper provides a detailed description of the proposed framework, including the failure-driven supervised fine-tuning procedure, capability-aware reinforcement learning objective, and inference-time help-seeking mechanism. Detailed implementation configurations, training hyperparameters, reward specifications, and optimization details are provided in Appendix A.2. The construction of training data, including failure trajectory collection, first critical error identification, and verified trajectory generation, is described in Sections 2.1 and 2.2, with additional details provided in the appendix. We provide descriptions of the evaluation benchmarks, comparison settings, and evaluation protocols in Section 3.1 and the supplementary materials.

REFERENCES

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.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Rühle, Laks V. S. Lakshmanan, and Ahmed Hassan Awadallah. Hybrid LLM: Cost-efficient and quality-aware query routing. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=02f3mUtqnM.

Peiyuan Feng, Yichen He, Guanhua Huang, Yuan Lin, Hanchong Zhang, Yuchen Zhang, and Hang Li. AGILE: A novel reinforcement learning framework of LLM agents. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=Ul3lDYo3XQ.

Bofei Gao, Feifan Song, Zhe Yang, Zefan Cai, Yibo Miao, Qingxiu Dong, Lei Li, Chenghao Ma, Liang Chen, Runxin Xu, Zhengyang Tang, Benyou Wang, Daoguang Zan, Shanghaoran Quan, Ge Zhang, Lei Sha, Yichang Zhang, Xuancheng Ren, Tianyu Liu, and Baobao Chang. Omni-MATH: A universal olympiad level mathematic benchmark for large language models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=yaqPf0KAlN.

Raghu Vamshi Hemadri, Humaira Firdowse Mohammed, Rishabh Maheshwary, Srivatsava Daruru, Sagar Davasam, Vikas Yadav, Srinivas Sunkara, and Sai Rajeswar. R2v agent: Teaching slms when to ask for help, 2026. URL https://arxiv.org/abs/2605.16604.

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.

Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. Constructing a multi-hop QA dataset for comprehensive evaluation of reasoning steps. In Donia Scott, Nuria Bel, and Chengqing Zong (eds.), *Proceedings of the 28th International Conference on Computational Linguistics*, pp. 6609–6625, Barcelona, Spain (Online), December 2020. International Committee on Computational Linguistics. doi: 10.18653/v1/2020.coling-main.580. URL https://aclanthology.org/2020.coling-main.580/.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: A benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:452–466, 2019. doi: 10.1162/tacl_a_00276. URL https://aclanthology.org/Q19-1026/.

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.

Wenjun Li, Dexun Li, Kuicai Dong, Cong Zhang, Hao Zhang, Weiwen Liu, Yasheng Wang, Ruiming Tang, and Yong Liu. Adaptive tool use in large language models with meta-cognition trigger. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13346–13370, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.655. URL https://aclanthology.org/2025.acl-long.655/.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E Gonzalez, M Kadous, and Ion Stoica. Routellm: Learning to route llms from preference data. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 34433–34448, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/5503a7c69d48a2f86fc00b3dc09de686-Paper-Conference.pdf.

Jing-Cheng Pang, Si-Hang Yang, Kaiyuan Li, Jiaji Zhang, Xiong-Hui Chen, Nan Tang, and Yang Yu. KALM: Knowledgeable agents by offline reinforcement learning from large language model rollouts. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=tb1MlJCY5g.

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP*

2023, pp. 5687–5711, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.378. URL https://aclanthology.org/2023.findings-emnlp.378/.

Yuxiao Qu, Tianjun Zhang, Naman Garg, and Aviral Kumar. Recursive introspection: Teaching LLM agents how to self-improve. In *ICML 2024 Workshop on Foundation Models in the Wild*, 2024. URL https://openreview.net/forum?id=g5wp1F3Dsr.

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 A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 68539–68551. Curran Associates, Inc., 2023. doi: 10.52202/075280-2997. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/d842425e4bf79ba039352da0f658a906-Paper-Conference.pdf.

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

Hao Sun, Jiayi Wu, Hengyi Cai, Xiaochi Wei, Yue Feng, Bo Wang, Shuaiqiang Wang, Yan Zhang, and Dawei Yin. AdaSwitch: Adaptive switching between small and large agents for effective cloud-local collaborative learning. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 8052–8062, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.458. URL https://aclanthology.org/2024.emnlp-main.458/.

Haoxiang Sun, Yingqian Min, Zhipeng Chen, Wayne Xin Zhao, and Ji-Rong Wen. Challenging the boundaries of reasoning: An olympiad-level math benchmark for large language models, 2026. URL https://arxiv.org/abs/2503.21380.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. MuSiQue: Multihop questions via single-hop question composition. *Transactions of the Association for Computational Linguistics*, 10:539–554, 2022. doi: 10.1162/tacl_a_00475. URL https://aclanthology.org/2022.tacl-1.31/.

Haozhe Wang, Long Li, Chao Qu, Weidi Xu, Fengming Zhu, Wei Chu, and Fangzhen Lin. To code or not to code? adaptive tool integration for math language models via expectation-maximization. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 3060–3075, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-256-5. doi: 10.18653/v1/2025.findings-acl.159. URL https://aclanthology.org/2025.findings-acl.159/.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents, 2024. URL https://arxiv.org/abs/2402.01030.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. ReST-MCTS*: LLM self-training via process reward guided tree search. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=8rcFOqEud5.

Wanjun Zhong, Ruixiang Cui, Yiduo Guo, Yaobo Liang, Shuai Lu, Yanlin Wang, Amin Saied, Weizhu Chen, and Nan Duan. AGIEval: A human-centric benchmark for evaluating foundation models. In Kevin Duh, Helena Gomez, and Steven Bethard (eds.), *Findings of the Association for Computational Linguistics: NAACL 2024*, pp. 2299–2314, Mexico City, Mexico, June 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-naacl.149. URL https://aclanthology.org/2024.findings-naacl.149/.

A Appendix

A.1 Inference and Evaluation Details

During inference, both the SLM and the assisting LLM use a temperature of 0.0. We report pass@1 across all evaluation benchmarks. For mathematical reasoning and QA, we adopt an LLM-as-a-judge evaluation protocol with Qwen3.7-Plus to assess answer correctness; HumanEval is evaluated using executable tests as described below. For QA tasks, we use `e5-base-v2` for semantic retrieval and employ Wikipedia 2018 as the shared knowledge base for both agents and the RAG system. The knowledge base is loaded onto the GPU using half precision.

A.2 Training Details

Model initialization. We use Qwen3-4B as the small language model (SLM) and Qwen3.6-27B-FP8 as the teacher and assisting large language model (LLM). BeaconAgent is initialized from the SLM after failure-driven supervised fine-tuning (SFT). The same LoRA adapter is further optimized during capability-aware reinforcement learning.

Failure-driven supervised fine-tuning. Following Section 2.1, we construct SFT demonstrations from failed SLM trajectories and teacher-verified repaired trajectories. The final SFT dataset contains 3,021 successful repaired trajectories, including 1,419 mathematical reasoning examples and 1,602 retrieval-augmented QA examples.

We fine-tune the SLM with LoRA for two epochs. The LoRA rank and scaling factor are set to 64 and 128, respectively. The SFT objective only optimizes assistant-generated tokens while excluding system prompts, user instructions, and tool observations.

Capability-aware GRPO training. For reinforcement learning, we construct a capability-aware training set containing 6,000 tasks, including 3,000 mathematical reasoning tasks and 3,000 retrieval-augmented QA tasks. The tasks are evenly distributed among four strata: $(\mathrm{math}, \mathrm{S})$, $(\mathrm{math}, \mathrm{H})$, $(\mathrm{QA}, \mathrm{S})$, and $(\mathrm{QA}, \mathrm{H})$.

For each task, we sample four no-help trajectories from the post-SFT policy to estimate the SLM capability boundary. A task is labeled as H if at least three trajectories produce incorrect final answers; otherwise, it is labeled as S. During GRPO training, dynamic relabeling is enabled. Specifically, a task can be relabeled from H to S when the retained rollout group size is sufficient and its estimated autonomous success rate exceeds the predefined threshold.

For GRPO optimization, each task generates up to eight rollout trajectories. We optimize the policy using a learning rate of 1×10^{-6} and a dual-clipped objective with $\epsilon = 0.2$ and $c_{\mathrm{dual}} = 3.0$. The advantage is computed by subtracting the mean reward within each rollout group without standard-deviation normalization. No KL regularization or entropy regularization is applied.

The complete optimization configurations are summarized in Table 5.

Reward design. The reward function consists of three components: (1) task outcome reward, (2) repeated-request penalty, and (3) critical-step alignment reward.

For a rollout i, the base reward follows Eq. equation 14. We characterize the reward design in terms of its relative preference structure, which specifies the intended ordering among autonomous

Table 5: Training configurations of BeaconAgent.

Configuration	Setting	Configuration	Setting
SLM backbone	Qwen3-4B	Teacher LLM	Qwen3.6-27B-FP8
SFT dataset size	3,021 trajectories	RL dataset size	6,000 tasks
RL domain split	Math 3,000 / QA 3,000	G_{target}	8
$G_{\min}$	2	τ_{rel}	0.375
SFT epochs	2	Learning rate	1×10^{-6}
Warmup ratio	0	LoRA rank / alpha	64 / 128
GRPO clipping ratio ϵ	0.20	Dual-clip coefficient	3.0
KL regularization	None	Entropy coefficient	0
Advantage normalization	Group mean subtraction only	Maximum agent steps	7
Partial rollout	Enabled	Weight synchronization	Every 4 local updates
Async staleness parameter	0.5		

execution, assistance seeking, task success, and auxiliary shaping signals. We do not claim that the reward configuration used in our experiments is uniquely optimal. Different deployment settings may adopt different preference strengths depending on the relative cost of external assistance and the desired level of intervention.

The reward assignment is conditioned on the estimated capability boundary. For within-capability tasks (S), the reward favors autonomous execution and discourages unnecessary assistance, whereas for beyond-capability tasks (H), the reward favors successful assisted execution, encouraging the SLM agent to seek assistance when the task exceeds its capability.

For within-capability tasks (S), the reward ordering is:

$$r_{0,1}^{\mathrm{S},u} > r_{1,1}^{\mathrm{S},u} > r_{0,0}^{\mathrm{S},u} > r_{1,0}^{\mathrm{S},u}. \tag{23}$$

Here, u denotes the task domain, z denotes final correctness, and the first subscript indicates whether help is requested. Thus, successful autonomous execution receives the highest reward, while unnecessary assistance is discouraged.

For beyond-capability tasks (H), the reward ordering is:

$$b_1^{\mathrm{H}} > r_{0,1}^{\mathrm{H}} > b_0^{\mathrm{H}} > r_{0,0}^{\mathrm{H}}. \tag{24}$$

Thus, successful assisted execution is preferred over autonomous execution on tasks beyond the SLM's capability, while unsuccessful trajectories remain penalized.

The repeated-request penalty λ_{rep} is set to 0.20 and is applied only after the first help request, discouraging redundant assistance while allowing necessary help seeking. The critical-step alignment reward assigns $+0.05$ when the first help request matches the reference first critical-error position identified during failure-driven supervision, and -0.05 otherwise. The alignment reward magnitude is intentionally smaller than the outcome reward scale, ensuring that task correctness remains the dominant optimization signal.

Before GRPO optimization, rewards are clipped using a success lower bound and a failure upper bound to maintain the separation between successful and failed trajectories:

$$R_{\min}^{+} > R_{\max}^{-}. \tag{25}$$

Implementation details. The GRPO training uses mixed batches containing mathematical reasoning and retrieval-augmented QA tasks from all capability categories. We use an asynchronous rollout–learner pipeline with partial rollout enabled. Policy optimization is applied only to SLM-generated tokens, while teacher-generated guidance and tool observations are excluded from the policy loss.

A.3 Evaluation Benchmarks

We evaluate BeaconAgent on eleven benchmarks spanning three domains: mathematical reasoning, retrieval-augmented question answering, and code generation. The detailed benchmark statistics and evaluation settings are described below.

Table 6: Evaluation benchmarks, dataset statistics, and evaluation settings across mathematical reasoning, retrieval-augmented question answering, and code generation.

Benchmark	Description and evaluation settings
Mathematical reasoning:	
MATH500	contains 500 challenging mathematical problems selected from the MATH benchmark and is directly used for evaluation.
AIME	consists of 90 problems from the 2022–2024 American Invitational Mathematics Examination (30 problems per year).
Gaokao2023	contains 385 mathematical problems collected from the 2023 Chinese National College Entrance Examination, the 2023 American Mathematics Competitions, and the 2023 American College Testing.
OlyMath	contains 200 olympiad-level mathematical problems designed to evaluate advanced mathematical reasoning.
Omni-MATH-500	contains 500 diverse mathematical problems covering multiple mathematical domains and difficulty levels.
Retrieval-augmented question answering:	
HotpotQA	is a multi-hop question answering benchmark that requires models to retrieve and combine information from multiple Wikipedia documents. Since the original test split contains a large number of instances, we randomly sample 500 questions for evaluation.
2WikiMultiHopQA (2Wiki)	is a large-scale multi-hop QA benchmark constructed from Wikipedia knowledge graphs, requiring models to perform multi-step reasoning across multiple evidence sources. We randomly sample 500 questions from its original test split for evaluation.
Bamboogle	is a challenging multi-hop question answering benchmark designed to evaluate evidence retrieval and compositional reasoning. It contains 125 questions, and we directly use the full benchmark for evaluation.
MuSiQue	is a multi-hop QA benchmark that requires models to combine multiple reasoning steps over retrieved evidence. We randomly sample 500 questions from its original test split for evaluation.
Natural Questions (NQ)	is an open-domain QA benchmark based on real user questions from Google search. We randomly sample 500 questions from its original test split for evaluation.
Together, these benchmarks evaluate the ability of BeaconAgent to utilize external evidence, perform multi-hop reasoning, and determine when additional assistance is required.	
Code generation:	
HumanEval:	We evaluate generalization beyond reasoning and QA tasks using **HumanEval**, which contains 164 Python programming problems. BeaconAgent generates a complete Python function through multi-step CodeAgent execution. The agent may test its code in a local interpreter but has no access to the official hidden assertions. After generation, the evaluator reconstructs the candidate source code, combines it with HumanEval's `check(candidate)` tests, and executes the resulting program in a separate Python subprocess. A candidate receives a score of 1 if all assertions pass and 0 otherwise. By default, we report candidate-level accuracy across all 164 problems; with `NUM_SAMPLES=1`, this metric approximately corresponds to pass@1.

A.4 Quality Control of Reference Trajectory Construction

In Section 2.2 (Capability-Aware Task Construction), we rerun the post-SFT SLM and the teacher LLM through the first critical error localization and minimal intervention with verified rollout procedure to construct one help-containing reference trajectory for each hard task. In a small fraction of cases, the initial teacher model fails to generate a trajectory that passes the final correctness verification after intervention.

To ensure the reliability of the reference trajectories used for RL alignment, we apply an additional quality-control procedure to these cases. Specifically, we replace the initial teacher model with a stronger model, Qwen3.8-Max, and rerun the same critical-error localization and minimal intervention procedure. The regenerated trajectory is retained only when it produces the correct final answer and satisfies the trajectory verification criteria.

After this quality-control step, each hard task is associated with one verified help-containing reference trajectory. The corresponding inserted `ASK_HELP` position t_x^* is obtained from the validated trajectory and is used as the reference critical-error position for subsequent capability-aware RL alignment.

A.5 LLM output-token usage for the three collaboration baselines and BeaconAgent

Table 7: LLM output-token usage for the three collaboration baselines and BeaconAgent. The assisted-trajectory mean conditions on receiving at least one LLM intervention.

Method	Total output tokens	Tokens / assisted trajectory	Avg.Acc (%)
AdaSwitch	382,825.04	403.00	47.80
R2vAgent	2,790,226.16	3,628.39	53.32
STEER	1,427,712.00	663.78	48.59
BeaconAgent	**116,115.89**	**49.39**	**57.55**

A.6 Definition of brief, targeted help.

In BeaconAgent, **brief, targeted help** refers to a minimal intervention from the teacher LLM that provides only the necessary guidance to correct or verify the SLM agent's current reasoning state, while leaving subsequent execution to the SLM agent. Specifically, the help is constrained to be **short**, **local**, and **actionable**: it does not contain the final answer, a complete solution, or a full derivation, but only addresses the identified failure point. Depending on the detected failure type, the teacher provides one of three forms of guidance: (1) PLAN, which suggests a next-step strategy without executing it; (2) VERIFY, which checks an intermediate result or evidence and identifies potential issues; and (3) REPAIR, which points out a local error and provides the corrected next step. After receiving the guidance, the SLM agent continues solving the task independently.

We implement minimal assistance through an explicit `ask_help` tool, which is registered as an available tool for the SLM agent. During execution, the SLM agent selects a help type (PLAN, VERIFY, or REPAIR) and submits a concise request describing its local uncertainty. The tool supplies the assisting LLM with the original problem, the preceding trajectory, the current help-seeking step, and the request. Type-specific prompts restrict the scope of assistance: PLAN suggests a next-step strategy without executing it; VERIFY identifies intermediate results, conditions, or evidence to check; and REPAIR provides guidance for correcting a local error. The prompts require brief, relevant guidance and prohibit final answers, complete solutions, and full derivations. For QA tasks, they additionally instruct the assisting LLM not to introduce outside knowledge. The response is returned as a tool observation, after which the SLM agent remains responsible for subsequent reasoning, tool use, and final answer generation. This design provides localized guidance without transferring task execution to the assisting LLM.

A.7 Definition of the First Critical Error

Definition of the First Critical Error

The first critical error is the earliest reasoning step in the trajectory where the small agent makes a genuine logical mistake that causes the subsequent reasoning process to deviate from a correct solution path.
A step should be identified as the first critical error only if all of the following conditions are satisfied:

1. **Causality:**
   The mistake directly contributes to the later failure. It is not merely an incorrect statement, minor imprecision, or harmless intermediate deviation.
2. **Earliest occurrence:**
   Among all erroneous steps, select the earliest step that introduces the failure-causing reasoning deviation. Do not select later errors that are only consequences or symptoms of the initial mistake.
3. **Recoverability:**
   The error can be corrected through one local help action without requiring the teacher to rewrite the entire solution strategy or restart the reasoning process.
4. **Reasoning-level failure:**
   The error must occur in the reasoning process, such as incorrect algebraic manipulation, invalid inference, wrong substitution, incorrect case analysis, flawed derivation, or inappropriate local transformation.

Do not consider a step a critical error if:

- it only reflects uncertainty, incomplete explanation, or lack of confidence;
- the reasoning is still valid and can potentially reach the correct answer;
- the error appears only after the trajectory has already deviated;
- it is only a formatting, notation, or presentation issue.

The goal is to identify the earliest point where a small amount of targeted assistance would prevent the trajectory from failing.

A.8 DEFINITIONS OF THREE TYPES OF HELP-SEEKING

ASK_HELP Type Definitions for math

Definitions:
1. verify
Use verify when the small agent has an intermediate result, evidence, calculation, or final answer that needs checking.
Typical signs:

- The path is mostly reasonable but the result may be wrong.
- The agent misreads the evidence from the previous step.
- The agent reaches a wrong final answer after plausible reasoning.
- Math answer is a decimal/format issue or a questionable intermediate result.

2. repair
Use repair when the agent has already gone down a broken local path and needs a local fix.
Typical signs:

- A specific mathematical step is wrong and should be corrected before continuing.
- The help should point out the local error and give the corrected next step.

3. plan
Use plan only when the main problem is the initial strategy, decomposition, or modeling choice.
Typical signs:

- The agent needs a better high-level solution strategy before continuing.
- The chosen brute-force/search/modeling strategy is inappropriate.
- A math problem needs a conceptual route, construction, case split, recurrence, or modeling plan.

Do not choose plan merely because the task is hard. In multi-step failed trajectories, plan should be relatively rare.
MATH-specific guidance:

- Do not use code execution errors as the reason to choose ASK_HELP(repair).
- If the trajectory contains code execution noise, ignore that noise when deciding the ask type.
- Focus only on reasoning logic: algebra, derivation, simplification, substitution, case split, modeling choice, or other math reasoning mistakes.
- Focus on reasoning logic, local algebra, symbolic manipulation, modeling choice, or evidence of a mistaken derivation.
- Do not delete, summarize, or rewrite the original trajectory content. Judge from the full raw trajectory as given.
- Choose the earliest recoverable failure step where a small help action would be useful.
- The target step must be the first genuine logic error that causes later drift, not a later symptom.
- Prefer the earliest step that a local help action could realistically repair.
- If the trajectory includes code execution errors, do not let them change the ask type by themselves.
- Math and QA are unrelated here; do not import any QA/search reasoning into this decision.

QA-specific ASK_HELP Type Definitions

Definitions:
1. verify
Use verify when the small agent has an intermediate evidence chain, candidate answer, or factual conclusion that needs checking.
2. repair
Use repair when the agent has followed a broken local path and needs a local correction.
Typical signs:

- Wrong search query or irrelevant evidence chain.
- A step that misuses retrieved information.
- A local reasoning mistake that should be corrected before continuing.

3. plan
Use plan only when the main failure is the initial decomposition or search strategy.
Typical signs:

- The agent needs a better multi-hop plan before continuing.
- It should decide which evidence to fetch or compare first.
- The current search strategy is fundamentally misdirected.

Do not choose plan merely because the task is hard.
QA-specific guidance:

- Do not use internal world knowledge to answer the question directly.
- Do not guess factual details from memory.
- Only judge the help step and ask type; do not rewrite the trajectory.
- Choose the earliest recoverable failure step where a small help action would be useful.

A.9 Example of a Help-Seeking Trajectory

The following example records a repair-type help interaction on a mathematics problem. After a Python execution error, the SLM requests targeted guidance, receives a brief hint, and resumes execution to obtain the answer $N = 4$. The transcript starts with the user message; the initial system prompt is omitted.

01 User

```
New task:
Cylinder $B$'s height is equal to the radius of cylinder $A$ and
  cylinder $B$'s radius is equal to the height $h$ of cylinder $A$.
  If the volume of cylinder $A$ is twice the volume of cylinder $B$,
  the volume of cylinder $A$ can be written as $N \pi h^3$ cubic
  units. What is the value of $N$?

[asy]
size(4cm,4cm);
path a=(0,-1.2)..(-10,0)--(10,0)..cycle;
path b=(17,-0.9)..(13,0)--(21,0)..cycle;
path c=(0,1.2)..(-10,0)--(10,0)..cycle;
path d=(17,0.9)..(13,0)--(21,0)..cycle;
draw(c);
draw(d);
draw(shift((0,7))*a);
draw(shift((0,7))*c);
draw(shift((0,15))*b);
draw(shift((0,15))*d);
draw((-10,0)--(-10,7));
draw((10,0)--(10,7));
draw((13,0)--(13,15));
draw((21,0)--(21,15));
draw(a);
draw(b);
label("$A$",(0,9),N);
label("$B$",(17,18),N);
[/asy]
```

02 SLM

````
Thought: I need to set up equations based on the given relationships
  between the dimensions of the cylinders and the fact that the
  volume of cylinder A is twice the volume of cylinder B. Then, I
  will solve for the value of $ N $ in the expression $ N \pi h^3 $.
Code:
```py
from sympy import symbols, Eq, solve

# Define variables
h = symbols('h')
r_A = symbols('r_A')
h_B = symbols('h_B')
r_B = symbols('r_B')

# Given relationships
# h_B = r_A
# r_B = h
# Volume of A = 2 * Volume of B
# Volume of a cylinder is πr^2h
# So, πr_A^2 * h_A = 2 * πr_B^2 * h_B

# Substitute the relationships into the volume equation
# πr_A^2 * h = 2 * πr_B^2 * h_B
````
````

02 SLM (continued)

```
# Since h_B = r_A and r_B = h, substitute these values
# πr_A^2 * h = 2 * πh^2 * r_A

# Simplify the equation
eq = Eq(π * r_A**2 * h, 2 * π * h**2 * r_A)

# Solve for r_A in terms of h
solution = solve(eq, r_A)
r_A_solution = solution[0]

# Now, substitute r_A back into the volume of A
# Volume of A = πr_A^2 * h
# Substitute r_A = h / 2
volume_A = π * (r_A_solution)**2 * h

# Simplify the expression for volume_A
volume_A_simplified = volume_A.simplify()

# The volume of A can be written as Nπh^3, so we need to find N
N = volume_A_simplified / (π * h**3)

final_answer(N)
```<end_code>

03 Tool call

```
Calling tools:
Call id: call_1
Type: function
Function: python_interpreter
Arguments:
from sympy import symbols, Eq, solve

# Define variables
h = symbols('h')
r_A = symbols('r_A')
h_B = symbols('h_B')
r_B = symbols('r_B')

# Given relationships
# h_B = r_A
# r_B = h
# Volume of A = 2 * Volume of B
# Volume of a cylinder is πr^2h
# So, πr_A^2 * h_A = 2 * πr_B^2 * h_B

# Substitute the relationships into the volume equation
# πr_A^2 * h = 2 * πr_B^2 * h_B
# Since h_B = r_A and r_B = h, substitute these values
# πr_A^2 * h = 2 * πh^2 * r_A

# Simplify the equation
eq = Eq(π * r_A**2 * h, 2 * π * h**2 * r_A)

# Solve for r_A in terms of h
solution = solve(eq, r_A)
r_A_solution = solution[0]

# Now, substitute r_A back into the volume of A
# Volume of A = πr_A^2 * h
# Substitute r_A = h / 2
```

03 Tool call (continued)

```
volume_A = π * (r_A_solution)**2 * h

# Simplify the expression for volume_A
volume_A_simplified = volume_A.simplify()

# The volume of A can be written as Nπh^3, so we need to find N
N = volume_A_simplified / (π * h**3)

final_answer(N)
```

04 Tool response

```
Call id: call_1
Error:
Code execution failed at line 'eq = Eq(π * r_A**2 * h, 2 * π * h**2 *
   r_A)' due to: InterpreterError: The variable `π` is not defined.
Now let's retry: take care not to repeat previous errors! If you have
   retried several times, try a completely different approach.
```

05 SLM — ASK_HELP(repair)

````
Thought: I should ask for limited help before continuing, because I
  might make a mistake by using `math.pi` with SymPy symbolic
  variables, which causes the equation to fail or simplify
  incorrectly. Therefore, I need to ask the larger model for limited
  help.
Code:
```py
# ASK_HELP(repair)
help_msg = ask_help(ask_type='repair', request='I might make a
  mistake by using `math.pi` with SymPy symbolic variables, which
  causes the equation to fail or simplify incorrectly. Therefore, I
  need to ask the larger model for limited help.')
print(help_msg)
```<end_code>
````

06 Tool call

```
Calling tools:
Call id: call_help_1
Type: function
Function: python_interpreter
Arguments:
# ASK_HELP(repair)
help_msg = ask_help(ask_type='repair', request='I might make a
  mistake by using `math.pi` with SymPy symbolic variables, which
  causes the equation to fail or simplify incorrectly. Therefore, I
  need to ask the larger model for limited help.')
print(help_msg)
```

07 Tool response — teacher guidance

```
Call id: call_help_1
Observation:
Execution logs:
Help:[HELP:repair] Use `sympy.pi` instead of `math.pi` or the
  undefined symbol `π` for symbolic algebra. This ensures SymPy
  treats pi as a symbolic constant rather than a float.<end_help>
```

07 Tool response — teacher guidance (continued)

```
Last output from code snippet:
None
```

08 SLM

```
Thought: The previous attempt failed because `π` was not defined. The
   help suggests using `sympy.pi` for symbolic algebra. I will now re
  -run the calculation using `sympy.pi` to solve for $N$.

Code:
```py
from sympy import symbols, Eq, solve, pi

# Define variables
h = symbols('h', positive=True)
r_A = symbols('r_A', positive=True)

# Given relationships:
# Cylinder A: radius = r_A, height = h
# Cylinder B: radius = h, height = r_A (since B's height = A's radius
  )
# Volume of A = 2 * Volume of B

# Volume formulas:
# V_A = pi * r_A^2 * h
# V_B = pi * h^2 * r_A

# Equation: V_A = 2 * V_B
eq = Eq(pi * r_A**2 * h, 2 * pi * h**2 * r_A)

# Solve for r_A in terms of h
solution = solve(eq, r_A)
# Filter for positive solution since dimensions must be positive
r_A_solution = [sol for sol in solution if sol.is_positive][0]

# Calculate Volume of A using the solved r_A
volume_A = pi * (r_A_solution)**2 * h

# The volume is given as N * pi * h^3
# So N = volume_A / (pi * h^3)
N = volume_A / (pi * h**3)

print(f"r_A in terms of h: {r_A_solution}")
print(f"Volume A: {volume_A}")
print(f"N: {N}")

final_answer(N)
```<end_code>
```

09 Tool call

```
Calling tools:
Call id: call_2
Type: function
Function: python_interpreter
Arguments:
from sympy import symbols, Eq, solve, pi

# Define variables
h = symbols('h', positive=True)
```

09 Tool call (continued)

```
r_A = symbols('r_A', positive=True)

# Given relationships:
# Cylinder A: radius = r_A, height = h
# Cylinder B: radius = h, height = r_A (since B's height = A's radius
  )
# Volume of A = 2 * Volume of B

# Volume formulas:
# V_A = pi * r_A^2 * h
# V_B = pi * h^2 * r_A

# Equation: V_A = 2 * V_B
eq = Eq(pi * r_A**2 * h, 2 * pi * h**2 * r_A)

# Solve for r_A in terms of h
solution = solve(eq, r_A)
# Filter for positive solution since dimensions must be positive
r_A_solution = [sol for sol in solution if sol.is_positive][0]

# Calculate Volume of A using the solved r_A
volume_A = pi * (r_A_solution)**2 * h

# The volume is given as N * pi * h^3
# So N = volume_A / (pi * h^3)
N = volume_A / (pi * h**3)

print(f"r_A in terms of h: {r_A_solution}")
print(f"Volume A: {volume_A}")
print(f"N: {N}")

final_answer(N)
```

10 Tool response — final result

```
Call id: call_2
Observation:
Execution logs:
r_A in terms of h: 2*h
Volume A: 4*pi*h**3
N: 4
Last output from code snippet:
4
```