AlphaApollo: A System for Deep Agentic Reasoning

Zhanke Zhou[1†] **Chentao Cao**[1*] **Xiao Feng**[1*] **Xuan Li**[1*] **Zongze Li**[1*] **Xiangyu Lu**[1*] **Jiangchao Yao**[3*] **Weikai Huang**[1] **Tian Cheng**[1] **Jianghangfan Zhang**[1] **Tangyu Jiang**[1] **Linrui Xu**[1] **Yiming Zheng**[1] **Brando Miranda**[4] **Tongliang Liu**[5,2] **Sanmi Koyejo**[4] **Masashi Sugiyama**[2,6] **Bo Han**[1,2]
[1]TMLR Group, Department of Computer Science, Hong Kong Baptist University; [2]RIKEN AIP;
[3]Cooperative Medianet Innovation Center, Shanghai Jiao Tong University; [4]Stanford University;
[5]Sydney AI Centre, The University of Sydney; [6]The University of Tokyo
[†]Team lead; [*]Equal contribution, listed in alphabetical order

Abstract

We present AlphaApollo[1], an agentic reasoning *system* that targets two bottlenecks in foundation-model reasoning: (1) limited reasoning capacity for complex, long-horizon problem solving and (2) unreliable test-time evolution without trustworthy verification. AlphaApollo orchestrates models and tools via three components: (i) *multi-turn agentic reasoning*, which formalizes model-environment interaction with structured tool calls and responses; (ii) *multi-turn agentic learning*, which applies turn-level reinforcement learning to optimize tool-use reasoning while decoupling actions from tool responses for stable training; and (iii) *multi-round agentic evolution*, which refines solutions through a propose–judge–update loop with tool-assisted verifications and long-horizon memory. Across seven math reasoning benchmarks and multiple model scales, AlphaApollo improves performance through reliable tool use ($>85\%$ tool-call success), substantial gains from multi-turn RL (Avg@32: Qwen2.5-1.5B-Instruct $1.07\% \rightarrow 9.64\%$, Qwen2.5-7B-Instruct $8.77\% \rightarrow 20.35\%$), and improvements from evolution (e.g., Qwen2.5-3B-Instruct $5.27\% \rightarrow 7.70\%$, Qwen2.5-14B-Instruct $16.53\% \rightarrow 21.08\%$). The code is available at `https://github.com/tmlr-group/AlphaApollo`.

1 Introduction

Foundation models (FMs) increasingly power diverse applications via explicit *reasoning*, decomposing complex tasks into manageable steps. Yet single-model reasoning remains insufficient for frontier problems or real-world tasks. As of December 2025, GPT-5 achieves 25.3% and Gemini 2.5 Pro 21.6% on Humanity's Last Exam (Phan et al., 2025), and 9.9% and 4.9% on ARC-AGI-2 (Chollet et al., 2025). Beyond math and code, limited compute and domain knowledge restrict performance in biology, chemistry, and healthcare, undermining reliability for scientific discovery.

Two bottlenecks limit FM reasoning are *(i) model-intrinsic capacity* to generate candidate solutions and *(ii) test-time evolution* to refine them. First, prompting and post-training are often used to enhance model capacity, but they largely depend on the base model's priors, making it unclear whether observed gains reflect *emergent* capabilities or mere elicitation from pre-training (e.g., step-by-step reasoning and self-reflection). Core reasoning skills (e.g., exact calculus and symbolic manipulation) remain constrained by next-token prediction (Yang et al., 2024c; Wang et al., 2025a). Second, without ground-truth verification, test-time evolution often relies on the model's own judgments, which can be subjective and unreliable (Gao et al., 2025). Finally, scalable parallel evolution remains under-explored, limiting efficiency and multi-model coordination, while long-horizon evolution cannot be achieved without an effective mechanism of memory.

This work introduces *AlphaApollo*, an agentic reasoning *system* designed to overcome these bottlenecks. Its design principle is to *orchestrate* models and tools into a *self-evolving system* for deep agentic reasoning (as in Fig. 1). AlphaApollo adopts a strategy of setting clear goals, concentrat-

[1]Project website: `https://alphaapollo.org/`

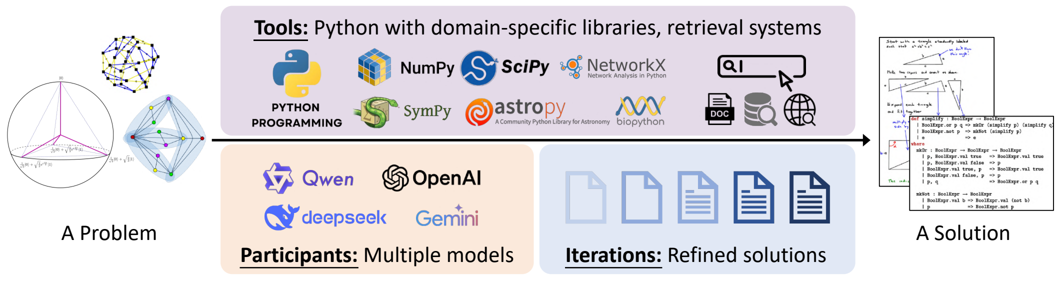

Tools: Python with domain-specific libraries, retrieval systems
PYTHON PROGRAMMING
NumPy
SciPy
NetworkX
SymPy
astropy
biopython
Qwen
OpenAI
deepseek
Gemini
Participants: Multiple models
Iterations: Refined solutions
A Problem
A Solution

Figure 1: An overview of the AlphaApollo System for problem-solving with foundation models.

ing expertise and resources, and coordinating systematic collaboration under organizational support, thereby enabling it to tackle complex problems. Specifically, AlphaApollo centers on three features to transcend the capacity limits of a single model:

- **Multi-turn agentic reasoning.** AlphaApollo solves tasks through iterative model-environment interaction: the model produces a structured action (tool call or answer), the environment executes tools and returns feedback, and the accumulated history serves as dynamic memory.
- **Multi-turn agentic learning.** AlphaApollo post-trains at the turn level, optimizing the model's *actions* (reasoning and tool invocation) while decoupling them from tool responses, which stabilizes RL/SFT and improves tool-use decisions (what to call, what to query, when to stop).
- **Multi-round agentic evolution.** AlphaApollo iteratively refined solutions with a propose-judge-update loop at test time, using tool-assisted checking and selective memory to refine candidates over multiple rounds. This evolution can coordinate multiple models in solving one problem.

Empirically, we evaluate AlphaApollo on seven mathematical reasoning benchmarks across multiple model scales and settings. AlphaApollo achieves consistent gains from agentic reasoning alone (e.g., for Avg@32, Qwen2.5-3B: $4.66\% \rightarrow 4.72\%$; Qwen2.5-14B: $10.82\% \rightarrow 13.49\%$), driven by reliable tool use with over 85% tool-call success across datasets. Multi-turn RL further boosts Avg@32 substantially (Qwen2.5-1.5B: $1.07\% \rightarrow 9.64\%$; Qwen2.5-3B: $4.72\% \rightarrow 13.35\%$; Qwen2.5-7B: $8.77\% \rightarrow 20.35\%$). Finally, test-time agentic evolution delivers additional scalable gains (Qwen2.5-3B: $4.79\% \rightarrow 6.92\%$; Qwen2.5-7B: $9.24\% \rightarrow 10.79\%$; Qwen2.5-14B: $16.53\% \rightarrow 21.08\%$), demonstrating reliable tool use, effective learning, and iterative self-improvement.

2 AlphaApollo

2.1 Multi-turn Agentic Reasoning

AlphaApollo structures the agentic reasoning as a *multi-turn interaction* between *model* and *environment* (Figure 2). In each turn, the model π_θ performs reasoning and invokes a tool call if needed, while the environment captures the tool call, executes the corresponding tool, and provides the tool responses back to the model. Specifically, a trajectory τ_t at t-th interacttion turn incorporates *prompt* p_t (the model's input), model *output* o_t, and environment *feedback* f_t, namely, $\tau_t = (p_t, o_t, f_t)$:

- The *prompt* p_t incorporates essential information from previous $t-1$ turns. For example, in a three-turn interaction, we have p_1 (the initial prompt), $p_2 = (p_1, o_1, f_1)$, and $p_3 = (p_2, o_2, f_2)$.
- The *output* o_t consists of the thinking process, followed by either a tool call or the final answer.
- The *feedback* f_t contains the tool response after executing the tool call.

This model-environment interaction terminates in two situations: (1) the model produces a final answer or (2) reaches the predefined maximum number of turns $T_{\max}$. We denote the number of interacted turns in a reasoning trajectory before termination as T ($T \leq T_{\max}$), and the final-turn trajectory as τ_T. Details of the model-environment synergies are introduced as follows.

Environment side. The environment is responsible for (1) hosting tools, (2) parsing the output o_t, (3) executing the tool call in o_t, and (4) returning the feedback f_t to the model. Specifically:

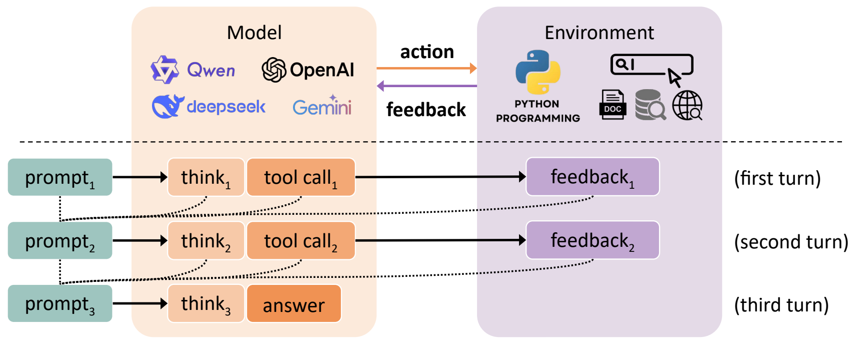


Figure 2: AlphaApollo's rollout is an iterative model-environment interaction. The model outputs an action, and the environment provides feedback. Each turn's trajectory is used as the prompt for the next turn, enabling dynamic memory. The final answer is produced when no tool calls are made.

- **Hosting tools.** Currently, AlphaApollo provides two types of tools: *computational tools* to solve precise calculations and *retrieval tools* to retrieve necessary knowledge for problem solving (detailed implementations are introduced in Appendix E). These tools are implemented as callable functions for the model, while users can easily plug in custom tools for extension.
- **Parsing outputs.** The environment parses the model's output o_t to extract: (1) *tool calls* wrapped within tool-specific tags (e.g., `<python_code>` and `</python_code>` for computational tools, `<local_rag>` and `</local_rag>` for retrieval tools) and (2) the *final answer* wrapped within `<answer>` and `</answer>`. The interaction terminates if the final answer is extracted.
- **Executing tools.** The system routes each tool call (parsed in the above step) to the corresponding tool hosting in the system. In AlphaApollo, each trajectory is handled by a separate environment, so tool calls across trajectories are executed *asynchronously*, enabling efficient parallelism.
- **Returning feedback.** The results of tool execution are wrapped within `<tool_response>` and `</tool_response>` tags and returned to the model as the feedback f_t.

Model side. Given prompt p_t, the model π_θ generates output o_t and receives environment feedback f_t. The system updates the memory and constructs the next-turn prompt p_{t+1}. Specifically:

- **Inference infrastructure.** AlphaApollo can generate the next token locally or remotely, with a particular model. The supported *inference backends* include vLLM (Kwon et al., 2023), SGLang (Zheng et al., 2024), HuggingFace Transformers (Huggingface, 2025), and external APIs (OpenAI, 2025). AlphaApollo employs Ray (Moritz et al., 2018) to parallelize trajectory generation. For a batch of questions, the system spawns multiple environments, each handling the interaction for one trajectory, and all environments run concurrently until all trajectories are collected.
- **Output generation.** The generated output o_t follows a structured format: it begins with a reasoning trajectory where the model articulates its thought process (enclosed within `<think>` and `</think>` tags), followed by a tool call or the final answer. This structured format aligns with the parsing patterns on the environment side, ensuring seamless model-environment interaction.
- **Memory management.** The prompt in each turn includes three types of information: the *task description* that specifies the user query and output format, *tool specifications* that describe the available tools and their usage formats, and *interaction history* that records previous outputs and feedback from earlier turns if available. Notably, AlphaApollo supports flexible memory strategies: by default, it concatenates all previous interactions as context, while for long-horizon tasks, it supports long-term memory that selectively retains high-quality trajectories (details in Sec. 2.3).

2.2 Multi-turn Agentic Learning

Based on the turn-level agentic reasoning system (Sec. 2.1) that decouples model-generated content from tool responses, AlphaApollo incorporates VeRL (Sheng et al., 2024) (an open-source framework for RL) into a *turn-level* optimization for stable and flexible agentic learning, along with versatile SFT/RL/parameter-efficient algorithms for versatile learning configurations.

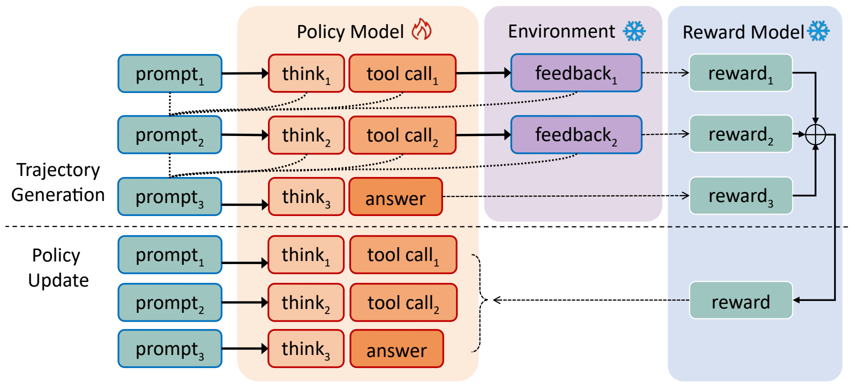


Figure 3: Illustration of multi-turn agentic RL in AlphaApollo. During generation, per-turn rewards are assigned based on model outputs and environment feedback, and summed to form the trajectory reward. During policy update, the policy is updated at each turn with non-model outputs masked.

Turn-level optimization. The full trajectory τ_T comprises multiple turns of model-generated content and the environment feedback. Notably, optimizing on the environment feedback can introduce instability (Jin et al., 2025; Feng et al., 2025). We discuss the trajectory-level optimization in Appendix F. For stable learning on the multi-turn interaction, AlphaApollo adopts turn-level optimization. This paradigm decouples model generations o_t from environment feedback f_t, thereby facilitating targeted optimization on o_t. For example, the turn-level GRPO is formulated as:

$$\mathcal{J}_{\text{Turn-GRPO}}(\theta)=\mathbb{E}_{(q,a)\sim\mathcal{D},\{o^{(i)}\}_{i=1}^{N}\sim\pi_{\text{old}}(\cdot|q)}\Bigg[\frac{1}{N}\sum_{i=1}^{N}\frac{1}{T^{(i)}}\sum_{t=1}^{T^{(i)}}\frac{1}{|o_t^{(i)}|}\sum_{k=1}^{|o_t^{(i)}|}$$

$$\min\left(\frac{\pi_\theta\left(o_{t,k}^{(i)}\mid p_t,o_{t,<k}^{(i)}\right)}{\pi_{\text{old}}\left(o_{t,k}^{(i)}\mid p_t,o_{t,<k}^{(i)}\right)}A_{t,k}^{(i)},\text{clip}\left(\frac{\pi_\theta\left(o_{t,k}^{(i)}\mid p_t,o_{t,<k}^{(i)}\right)}{\pi_{\text{old}}\left(o_{t,k}^{(i)}\mid p_t,o_{t,<k}^{(i)}\right)},1-\epsilon,1+\epsilon\right)A_{t,k}^{(i)}\right)-\beta\mathbb{D}_{\text{KL}}\left[\pi_\theta\|\pi_{\text{ref}}\right]\Bigg],$$

where $o_{t,k}^{(i)}$ denotes k-th token of the model's output in t-th turn of the i-th trajectory, N is the number of group samples for GRPO, $A_{t,k}^{(i)}=\frac{r(\tau_{T^{(i)}})-\bar{r}}{\sigma_r}$ where r is the reward by evaluating the answer proposed in turn T, $\bar{r}$ and σ_r are the baseline and the normalization across all N trajectories of one question. A indicates the advantage and ϵ is the clipping bound, and $\beta\,\mathbb{D}_{\text{KL}}[\pi_\theta\,\|\,\pi_{\text{ref}}]$ is the KL divergence term using an unbiased estimator (Schulman, 2020) weighted by β. Here, token importance ratios $\pi_\theta(o_{t,k}^{(i)}|p_t,o_{t,<k}^{(i)})/\pi_{\text{old}}(o_{t,k}^{(i)}|p_t,o_{t,<k}^{(i)})$ are computed conditioned on p_t and generated tokens $o_{t,<k}$, which incorporate the query and prior turns' generations and tool responses.

Similarly, the turn-level SFT is formulated as:

$$\mathcal{J}_{\text{Turn-SFT}}(\theta)=\mathbb{E}_{\tau_T\sim\mathcal{D}_{\text{SFT}}}\left[\sum_{t=1}^{T}\sum_{k=1}^{|o_t|}\log\pi_\theta(o_{t,k}\mid p_t,o_{t,<k})\right], \tag{1}$$

where τ_T is a full trajectory curated from the multi-turn agentic reasoning (Sec. 2.1).

Post-training features. To create a versatile and user-friendly agentic learning system, AlphaApollo incorporates the following post-training features:

- **Algorithms:** AlphaApollo supports common SFT/RL algorithms for post-training, such as PPO (Schulman et al., 2017), GRPO (Shao et al., 2024), and DAPO (Yu et al., 2025).
- **Models:** AlphaApollo enables agentic learning across various foundation model families, including Qwen2.5 (Yang et al., 2024a), Qwen3 (Yang et al., 2025), and Llama3.2 (Grattafiori et al., 2024). Additionally, it provides a multi-modal preprocessing and training pipeline for post-training with multi-modal foundation models, such as Qwen2.5-VL families (Bai et al., 2025).
- **Flexible advantage estimation:** AlphaApollo supports both trajectory- and turn-level advantage estimation in RL, making it compatible with emerging process reward modeling algorithms.

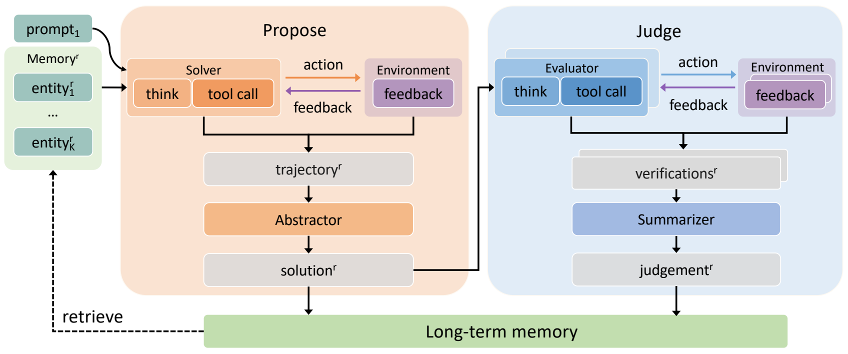


Figure 4: Illustration of multi-round agentic evolution in AlphaApollo. The model iteratively refines its strategies through a propose-judge-update evolutionary loop. A long-term memory is introduced to prevent future errors and promote efficient strategies in subsequent rounds.

- **Parameter-efficient fine-turning:** Leveraging FSDP and vLLM as backends, AlphaApollo enables LoRA (Hu et al., 2022) for efficient post-training of large-scale models.

2.3 Multi-round Agentic Evolution

AlphaApollo employs a test-time evolution mechanism to iteratively refine solutions (Figure 4). Operating via a propose-judge-update loop, the system coordinates multiple agents to generate solutions, evaluate them, and store the judgment in long-term memory to guide subsequent rounds.

Overview. We denote the system's initial input as p_1 and the long-term memory as $\mathcal{M}$, initialized as $\mathcal{M}^{(1)} = \emptyset$. The process iterates over multiple rounds $r = \{1, \dots, R\}$. In each round, the system coordinates four specialized agents π to instantiate the pipeline, yielding a *candidate solution* $\hat{s}^{(r)}$ and *diagnostic judgment* $j^{(r)}$. These outputs are formed as a memory entity $(\hat{s}^{(r)}, j^{(r)})$, which is committed to $\mathcal{M}$. The $\mathcal{M}^{(r+1)}$ is retrieved from $\mathcal{M}$ to condition the policy π for the subsequent evolution. Detailed prompts and implementations can be found in Appendix G.2. The pipeline is formulated as follows:

$$\begin{aligned} \textbf{1. Propose:} &\quad \tau^{(r)} \sim \pi_{\text{solver}}(\cdot \mid p_1, \mathcal{M}^{(r)}), \quad \hat{s}^{(r)} \sim \pi_{\text{abs}}(\tau^{(r)}); \\ \textbf{2. Reflection:} &\quad \mathcal{V}^{(r)} = \{v_i\}_{i=1}^{N} \sim \pi_{\text{eval}}(\cdot \mid \hat{s}^{(r)}), \quad j^{(r)} \sim \pi_{\text{sum}}(\mathcal{V}^{(r)}); \\ \textbf{3. Update:} &\quad \mathcal{M} = \mathcal{M} \cup \{(\hat{s}^{(r)}, j^{(r)})\}, \quad \mathcal{M}^{(r+1)} = \text{Retrieve}(\mathcal{M}). \end{aligned} \tag{2}$$

We detail the role of each agent in this pipeline below.

- `Solver` (π_{solver}): *Given a problem and memory state, generates a multi-turn reasoning trajectory.* The solver interacts with the environment to generate a complete trajectory $\tau^{(r)}$.
- `Abstractor` (π_{abs}): *Given $\tau^{(r)}$, generates a condensed solution.* The abstractor compresses $\tau^{(r)}$ into a condensed solution $\hat{s}^{(r)}$ to fit the context window while preserving essential information.
- `Evaluator` (π_{eval}): *Given $\hat{s}^{(r)}$, generates verifications $\mathcal{V}^{(r)}$.* For deterministic tasks, it integrates execution feedback. For open-ended tasks, it aggregates N sampled critiques via *majority voting*.
- `Summarizer` (π_{sum}): *Given $\mathcal{V}^{(r)}$, generates comprehensive judgement $j^{(r)}$.* Synthesize $\mathcal{V}^{(r)}$ into a high-level judgment $j^{(r)}$, remove redundancy to provide clean advice for π_{solver} in future rounds.

Long-term memory for evolution. To enable long-horizon evolution, the memory module supports efficient retrieval to prevent excessive context, including the overlong content of correctness and solution. Specifically, the long-term memory operates in two phases:

- **Store:** At the end of round r, the system transitions the memory state by appending the new entity: $\mathcal{M} \cup \{(\hat{s}^{(r)}, j^{(r)})\}$. This accumulation ensures the history of exploration is preserved.
- **Retrieval:** To yield $\mathcal{M}^{(r+1)}$, the system selects the Top-K entries from the memory under a scoring function prioritizing (1) the correct solution and (2) shorter solutions in correct groups.

Table 1: Agentic reasoning results (Avg@32/Pass@32 in %). Base is evaluated without training and tools; AlphaApollo is evaluated with tools enabled without training. **Bold** marks the better results.

Dataset	**Qwen2.5-3B-Instruct**		**Qwen2.5-7B-Instruct**		**Qwen2.5-14B-Instruct**	
	Base	**AlphaApollo**	**Base**	**AlphaApollo**	**Base**	**AlphaApollo**
AIME24	5.21 / 26.67	**5.52 / 30.00**	**12.19** / 36.67	8.85 / **56.67**	13.44 / 46.67	**16.98/ 60.00**
AIME25	**3.23 / 36.67**	2.19 / 23.33	**8.23** / 36.67	6.15 / **36.67**	**12.29** / 43.33	11.77 / **46.67**
CMIMC25	1.17 / 17.50	**3.36 / 30.00**	4.02 / 30.00	**7.42 / 40.00**	4.61 / 27.50	**11.48 / 40.00**
HMMT25 Feb	0.52 / 10.00	**2.60 / 16.67**	2.08 / 23.33	**6.67 / 33.33**	3.23 / 23.33	**9.58 / 40.00**
HMMT25 Nov	**3.02 / 20.00**	2.50 / 23.33	5.00 / 23.33	**6.04 / 26.67**	5.73 / 20.00	**7.29 / 23.33**
BRUMO25	**11.25** / 40.00	8.44 / **46.67**	**18.23** / 50.00	17.18 / **50.00**	22.29 / 43.33	**22.60 / 60.00**
SMT 2025	8.25 / 32.08	**8.43 / 41.51**	**11.62** / 39.62	9.08 / **41.51**	14.15 / 41.51	**14.74 / 49.06**
Average	4.66 / 26.13	**4.72 / 30.22** (0.06↑) / (4.09↑)	8.77 / 34.23	**8.77 / 40.69** (0.00↑) / (6.46↑)	10.82 / 35.10	**13.49 / 45.58** (2.67↑) / (10.48↑)

Table 2: Agentic learning results (Avg@32 in %) for AlphaApollo. No-training evaluates AlphaApollo using tools without training. +LE (MATH-LightEval (Hendrycks et al., 2021)), +LIMR (Li et al., 2025), and +DS (DeepScaleR (Luo et al., 2025)) denote training the model on the corresponding datasets. **Bold** marks the best performance.

Dataset	**Qwen2.5-1.5B-Instruct**				**Qwen2.5-3B-Instruct**				**Qwen2.5-7B-Instruct**			
	No-training	**AlphaApollo (training)**			**No-training**	**AlphaApollo (training)**			**No-training**	**AlphaApollo (training)**		
		+LE	+LIMR	+DS		+LE	+LIMR	+DS		+LE	+LIMR	+DS
AIME24	0.63	3.77	3.74	**8.96**	5.52	9.14	14.35	**20.92**	8.85	22.91	19.40	**25.50**
AIME25	0.73	2.68	**15.36**	14.67	2.19	13.06	**14.14**	9.33	6.15	16.58	15.28	**17.58**
CMIMC25	0.63	5.25	**7.78**	7.11	3.36	6.32	7.80	**10.58**	7.42	13.16	14.14	**16.97**
HMMT25 Feb	1.35	4.47	**7.27**	6.51	2.60	12.27	12.61	**14.07**	6.67	13.33	9.21	**18.30**
HMMT25 Nov	0.83	1.73	**5.75**	4.93	2.50	**6.48**	5.29	4.21	6.04	11.21	12.28	**13.11**
BRUMO25	1.98	8.77	**15.96**	14.98	8.44	**23.22**	21.30	21.62	17.18	30.26	27.70	**33.10**
SMT 2025	1.36	5.83	**12.07**	10.31	8.43	13.11	**14.60**	10.72	9.08	16.65	**18.00**	17.88
Average	1.07	4.64 (3.57↑)	**9.70** (8.63↑)	9.64 (8.57↑)	4.72	11.94 (7.22↑)	12.87 (8.15↑)	**13.35** (8.33↑)	8.77	17.73 (8.96↑)	16.57 (7.80↑)	**20.35** (11.58↑)

Table 3: Agentic evolution results (accuracy in %). w/o Evo uses tools without evolution; +Evo enables agentic evolution with tools. **Bold** marks the better result.

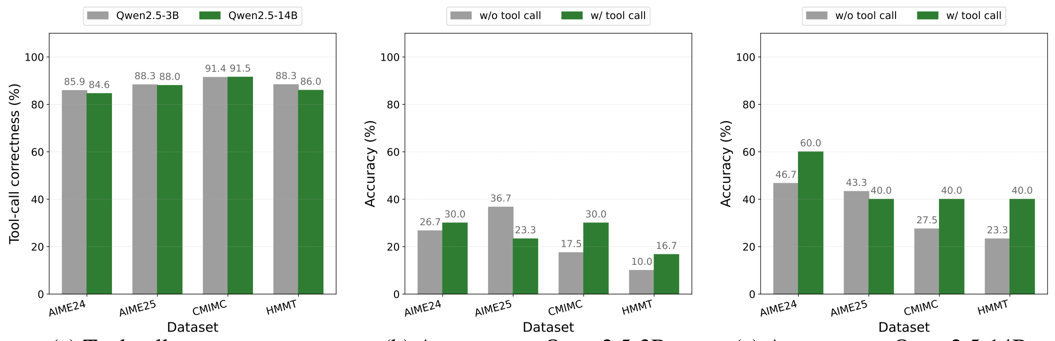
Datasets	**Qwen2.5-3B-Instruct**		**Qwen2.5-7B-Instruct**		**Qwen2.5-14B-Instruct**	
	w/o Evo	**+Evo**	**w/o Evo**	**+Evo**	**w/o Evo**	**+Evo**
AIME24	6.67	**10.00**	11.67	**12.50**	19.17	**23.33**
AIME25	**7.50**	**7.50**	8.33	**11.67**	18.33	**24.17**
CMIMC 25	3.12	**6.25**	10.00	**12.50**	15.00	**19.38**
HMMT25 Feb	3.33	**5.00**	10.00	**10.83**	15.83	**20.00**
HMMT25 Nov	1.67	**4.17**	**7.50**	**7.50**	7.50	**12.50**
BRUMO25	7.50	**12.50**	11.67	**17.50**	26.67	**31.67**
SMT 2025	7.08	**8.49**	8.02	**8.96**	13.21	**16.51**
Average	5.27	**7.70** (2.43↑)	9.60	**11.64** (2.04↑)	16.53	**21.08** (4.55↑)

Retrieving high-level judgment j promotes promising strategies while preventing recurrent errors. Furthermore, the module employs thread-safe locks to decouple storage from execution, enabling asynchronous agents to reliably synchronize insights regardless of latency.

Parallel Evolution. To accelerate reasoning, AlphaApollo distributes the evolution pipeline across multiple worker threads. This architecture supports heterogeneous solvers (e.g., mixing open-source models with proprietary APIs or varying sampling temperatures). Agents synchronize via the shared long-term memory: when one solver commits a high-quality solution, it becomes available to others, fostering collective intelligence that guides the evolution toward a better solution.

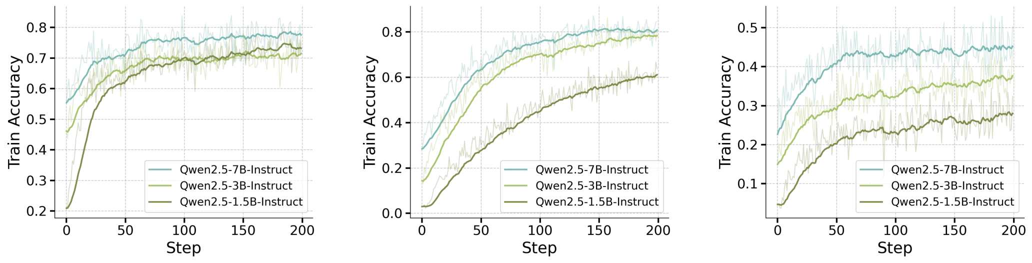
3 Experiments

Experiment setup. We evaluate AlphaApollo across various models, from Qwen2.5-1.5B-Insturct to Qwen2.5-14B-Instruct. We evaluate on AIME24, AIME25, CMIMC, HMMT, BRUMO, and SMT. We set a maximum of 4 interaction rounds. Details are in Appendix E, F, and G.



(a) Tool-call success rate. (b) Accuracy on Qwen2.5-3B. (c) Accuracy on Qwen2.5-14B.

Figure 5: Tool-call performance and accuracy enhancements in AlphaApollo using Qwen models.



(a) MATH-LightEval (b) LIMR (c) DeepScaleR

Figure 6: Training accuracy curves of experiments in Table.2.

Main results. We evaluate AlphaApollo from three aspects: agentic reasoning, agentic learning, and agentic evolution. The corresponding results are summarized in Tables 1, 2, and 3, respectively.

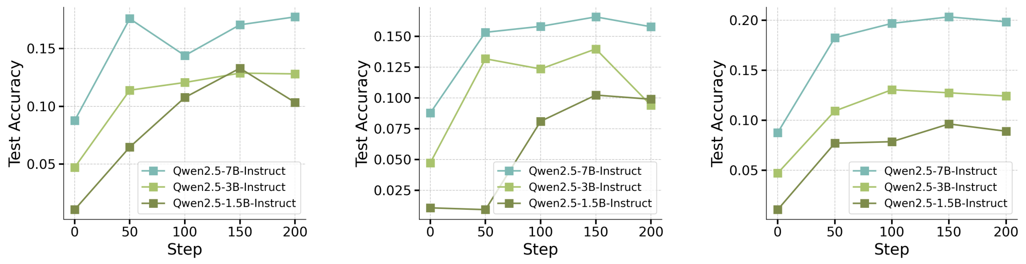
Agentic reasoning with tools enables consistent performance gain across model scales. In Table 1, Figures 5b, and 5c, AlphaApollo consistently outperforms no-tool versions across all models, demonstrating improvements from 4.66% to 4.72% on Qwen2.5-3B-Instruct and from 10.82% to 13.49% on Qwen2.5-14B-Instruct. The observed gains in reasoning capability primarily stem from improved tool-call reliability. In Figure 5a, AlphaApollo's model-friendly tool-calling module achieves a success rate exceeding 85% across all evaluated datasets. These demonstrate that AlphaApollo consistently excels in boosting FMs' capabilities, both in reasoning and tool utilization.

Agentic learning in AlphaApollo substantially improves reasoning across model scales. As shown in Table 2, training on DeepScaleR consistently yields large gains over tool-augmented baselines without training, improving average accuracy by +8.57 on Qwen2.5-1.5B-Instruct, +8.33 on Qwen2.5-3B-Instruct (MATH-LightEval, from 4.72% to 13.35%), and +11.58 on Qwen2.5-7B-Instruct, demonstrating strong scalability of agentic reasoning improvements. Figures 6 and 7 further show steady performance growth on both training and test sets. We additionally compare full-parameter and LoRA training (Figure 8). Full-parameter optimization exhibits faster learning dynamics, with quicker accuracy gains, reduced entropy loss, and consistently higher final performance, indicating the advantage of full-capacity adaptation for agentic reasoning.

Agentic evolution further amplifies reasoning capability through iterative self-improvement. In Table 3, enabling evolution (+Evo) improves over the non-evolving baseline (w/o Evo). The strongest gains occur on Qwen2.5-14B-Instruct, where average accuracy increases from 16.53% to 21.08% (+4.55), including large boosts on AIME24 and AIME25. Improvements also persist on other backbones, with average gains of +2.43 on Qwen2.5-3B-Instruct. Figure 9 reveals a stable upward trend across evolving rounds with clear scale-dependent gaps, indicating that agentic evolution reliably enhances reasoning performance and scales the benefit with model capability.

Overall, these results show that AlphaApollo delivers consistent, scalable gains by unifying agentic reasoning, learning, and evolution, enabling reliable tool use and iterative refinement of reasoning.

Case studies. Under AlphaApollo, models exhibit diverse cognitive reasoning behaviors, rooted in their pre-training data and enabling deliberate reasoning. We showcase in Figure 30 and present complete case studies in Appendix H.2 while summarizing following agentic reasoning behaviors:



(a) MATH-LightEval (b) LIMR (c) DeepScaleR

Figure 7: Validation accuracy curves of experiments in Table.2.

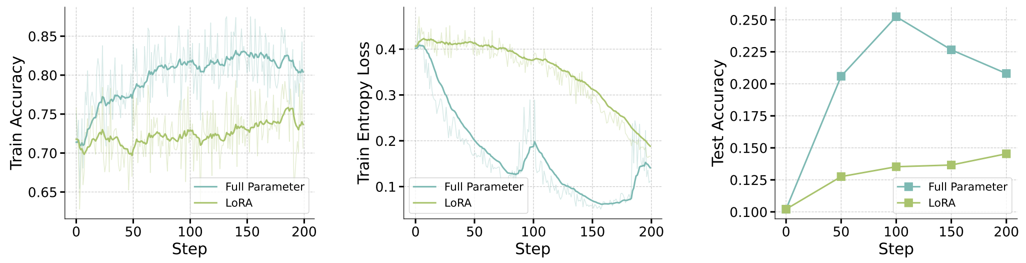


Figure 8: Full parameters vs. LoRA training on Qwen2.5-14B-Instruct with MATH-LightEval.

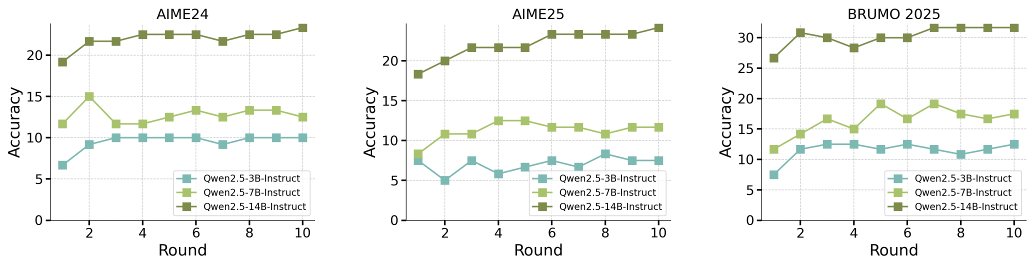


Figure 9: Accuracy across successive self-evolution rounds for different model scales.

- **Decomposition**. The model breaks down a complex problem into smaller, more manageable sub-problems. This strategy not only reduces cognitive load but also increases the likelihood of solving each component correctly, which in turn contributes to the accuracy of the final solution.
- **Correction**. During the reasoning process, the model frequently identifies potential mistakes in intermediate steps and revises them. Such self-corrective behavior shows that the model can refine its outputs dynamically rather than strictly following an error-prone initial trajectory.
- **Verification**. The model actively checks intermediate results against either external tools or internal consistency rules. This verification step functions as a safeguard, filtering out unreasonable solutions and ensuring that the final answer is logically sound.
- **Backtracking**. When encountering contradictions, the model is capable of retracing earlier steps and exploring alternative reasoning paths. This behavior resembles human-like problem solving, where a failed attempt triggers a systematic search for better strategies.

4 Conclusion

We presented AlphaApollo, a self-evolving agentic reasoning system that addresses limited long-horizon reasoning capacity and unreliable test-time refinement by orchestrating models and tools. AlphaApollo combines multi-turn agentic reasoning, turn-level agentic learning, and multi-round agentic evolution to enable structured tool use, stable RL optimization, and iterative self-improvement with tool-assisted verifications and long-horizon memory. Across seven math reasoning benchmarks and multiple model scales, AlphaApollo delivers consistent gains through reliable tool use, multi-turn reinforcement learning, and further improvements from test-time evolution.

REFERENCES

Saaket Agashe, Yue Fan, Anthony Reyna, and Xin Eric Wang. Llm-coordination: evaluating and analyzing multi-agent coordination abilities in large language models. *arXiv preprint arXiv:2310.03903*, 2023.

Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. Self-RAG: Learning to retrieve, generate, and critique through self-reflection. In *ICLR*, 2024.

Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, et al. Qwen2. 5-vl technical report. *arXiv preprint arXiv:2502.13923*, 2025.

Jiajun Chai, Guojun Yin, Zekun Xu, Chuhuai Yue, Yi Jia, Siyu Xia, Xiaohan Wang, Jiwen Jiang, Xiaoguang Li, Chengqi Dong, et al. Rlfactory: A plug-and-play reinforcement learning post-training framework for llm multi-turn tool-use. *arXiv preprint arXiv:2509.06980*, 2025a.

Jingyi Chai, Shuo Tang, Rui Ye, Yuwen Du, Xinyu Zhu, Mengcheng Zhou, Yanfeng Wang, Weinan E, Yuzhi Zhang, Linfeng Zhang, and Siheng Chen. Scimaster: Towards general-purpose scientific ai agents, part i. x-master as foundation: Can we lead on humanity's last exam? *arXiv preprint arXiv:2507.05241*, 2025b.

Kaiyan Chang, Yonghao Shi, Chenglong Wang, Hang Zhou, Chi Hu, Xiaoqian Liu, Yingfeng Luo, Yuan Ge, Tong Xiao, and Jingbo Zhu. Step-level verifier-guided hybrid test-time scaling for large language models. *arXiv preprint arXiv:2507.15512*, 2025.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. Fireact: Toward language agent fine-tuning. *arXiv preprint arXiv:2310.05915*, 2023a.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*, 2023b.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023c.

Francois Chollet, Mike Knoop, Gregory Kamradt, Bryan Landers, and Henry Pinkard. Arc-agi-2: A new challenge for frontier ai reasoning systems. *arXiv preprint arXiv:2505.11831*, 2025.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *ICML*, 2024.

Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024.

Lang Feng, Zhenghai Xue, Tingcong Liu, and Bo An. Group-in-group policy optimization for llm agent training. In *NeurIPS*, 2025.

Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, et al. A survey of self-evolving agents: On path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*, 2025.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *ICML*, 2023a.

Shen Gao, Zhengliang Shi, Minghang Zhu, Bowen Fang, Xin Xin, Pengjie Ren, Zhumin Chen, Jun Ma, and Zhaochun Ren. Confucius: Iterative tool learning from introspection feedback by easy-to-difficult curriculum. In *AAAI*, 2024.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2023b.

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.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 2020.

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 *NeurIPS*, 2021.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *ICLR*, 2024.

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 *ICLR*, 2022.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *ICLR*, 2024.

Huggingface. Huggingface transformers, 2025. URL `https://huggingface.co/docs/transformers`.

Dongfu Jiang, Yi Lu, Zhuofeng Li, Zhiheng Lyu, Ping Nie, Haozhe Wang, Alex Su, Hui Chen, Kai Zou, Chao Du, et al. Verltool: Towards holistic agentic reinforcement learning with tool use. *arXiv preprint arXiv:2509.01055*, 2025.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. In *NeurIPS*, 2023.

Tim Knappe, Ryan Li, Ayush Chauhan, Kaylee Chhua, Kevin Zhu, and Sean O'Brien. Semantic self-consistency: Enhancing language model reasoning via semantic weighting. *arXiv preprint arXiv:2410.07839*, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *SOSP*, 2023.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *NeurIPS*, 2020.

Xuefeng Li, Haoyang Zou, and Pengfei Liu. Limr: Less is more for rl scaling. *arXiv preprint arXiv:2502.11886*, 2025.

Yunxuan Li, Yibing Du, Jiageng Zhang, Le Hou, Peter Grabowski, Yeqing Li, and Eugene Ie. Improving multi-agent debate with sparse communication topology. *arXiv preprint arXiv:2406.11776*, 2024.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh Murthy, Liangwei Yang, Silvio Savarese, Juan Carlos Niebles, Huan Wang, Shelby Heinecke, and Caiming Xiong. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. *arXiv preprint arXiv:2406.18518*, 2024.

Pan Lu, Bowen Chen, Sheng Liu, Rahul Thapa, Joseph Boen, and James Zou. Octotools: An agentic framework with extensible tools for complex reasoning. *arXiv preprint arXiv:2502.11271*, 2025.

Michael Luo, Sijun Tan, Justin Wong, Xiaoxiang Shi, William Tang, Manan Roongta, Colin Cai, Jeffrey Luo, Tianjun Zhang, Erran Li, Raluca Ada Popa, and Ion Stoica. Deepscaler: Surpassing o1-preview with a 1.5b model by scaling rl, 2025. Notion Blog.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. In *NeurIPS*, 2023.

Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. Sympy: symbolic computing in python. *PeerJ Computer Science*, 2017.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *OSDI*, 2018.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*, 2025.

Ranjita Naik, Varun Chandrasekaran, Mert Yuksekgonul, Hamid Palangi, and Besmira Nushi. Diversity of thought improves reasoning abilities of llms. *arXiv preprint arXiv:2310.07088*, 2023.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenAI. Openai api platform, 2025. URL `https://platform.openai.com`.

Debjit Paul, Mete Ismayilzada, Maxime Peyrard, Beatriz Borges, Antoine Bosselut, Robert West, and Boi Faltings. Refiner: Reasoning feedback on intermediate representations. *arXiv preprint arXiv:2304.01904*, 2023.

Long Phan, Alice Gatti, Ziwen Han, Nathaniel Li, Josephina Hu, Hugh Zhang, Chen Bo Calvin Zhang, Mohamed Shaaban, John Ling, Sean Shi, et al. Humanity's last exam. *arXiv preprint arXiv:2501.14249*, 2025.

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. *arXiv preprint arXiv:2210.03350*, 2022.

Jianing Qi, Xi Ye, Hao Tang, Zhigang Zhu, and Eunsol Choi. Learning to reason across parallel samples for llm reasoning. *arXiv preprint arXiv:2506.09014*, 2025.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *ICLR*, 2024.

Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, et al. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *arXiv preprint arXiv:2505.20286*, 2025.

Yuxiao Qu, Tianjun Zhang, Naman Garg, and Aviral Kumar. Recursive introspection: Teaching language model agents how to self-improve. In *NeurIPS*, 2024.

Gollam Rabby, Farhana Keya, and Sören Auer. Mc-nest: Enhancing mathematical reasoning in large language models leveraging a monte carlo self-refine tree. *arXiv preprint arXiv:2411.15645*, 2024.

John Schulman. Approximating kl divergence, 2020. URL `http://joschu.net/blog/kl-approx.html`.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Ning Shang, Yifei Liu, Yi Zhu, Li Lyna Zhang, Weijiang Xu, Xinyu Guan, Buze Zhang, Bingcheng Dong, Xudong Zhou, Bowen Zhang, et al. rstar2-agent: Agentic reasoning technical report. *arXiv preprint arXiv:2508.20722*, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Weizhou Shen, Chenliang Li, Hongzhan Chen, Ming Yan, Xiaojun Quan, Hehong Chen, Ji Zhang, and Fei Huang. Small llms are weak tool learners: A multi-llm agent. In *EMNLP*, 2024.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. In *NeurIPS*, 2023.

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.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

Vighnesh Subramaniam, Yilun Du, Joshua B. Tenenbaum, Antonio Torralba, Shuang Li, and Igor Mordatch. Multiagent finetuning: Self improvement with diverse reasoning chains. *arXiv preprint arXiv:2501.05707*, 2025.

Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. In *ICCV*, 2023.

Amir Taubenfeld, Tom Sheffer, Eran Ofek, Amir Feder, Ariel Goldstein, Zorik Gekhman, and Gal Yona. Confidence improves self-consistency in llms. *arXiv preprint arXiv:2502.06233*, 2025.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, et al. Scipy 1.0: Fundamental algorithms for scientific computing in python. *Nature Methods*, 2020.

Guan Wang, Jin Li, Yuhao Sun, Xing Chen, Changling Liu, Yue Wu, Meng Lu, Sen Song, and Yasin Abbasi Yadkori. Hierarchical reasoning model. *arXiv preprint arXiv:2506.21734*, 2025a.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *ICLR*, 2022.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, et al. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025b.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *COLM*, 2024.

Zhenghai Xue, Longtao Zheng, Qian Liu, Yingru Li, Xiaosen Zheng, Zejun Ma, and Bo An. Simpletir: End-to-end reinforcement learning for multi-turn tool-integrated reasoning. *arXiv preprint arXiv:2509.02479*, 2025.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024a.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *ICLR*, 2024b.

Haotong Yang, Yi Hu, Shijia Kang, Zhouchen Lin, and Muhan Zhang. Number cookbook: Number understanding of language models and how to improve it. *arXiv preprint arXiv:2411.03766*, 2024c.

Ori Yoran, Tomer Wolfson, Ben Bogin, Uri Katz, Daniel Deutch, and Jonathan Berant. Answering questions by meta-reasoning over multiple chains of thought. In *EMNLP*, 2023.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative ai by backpropagating language model feedback. *Nature*, 2025.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. Agenttuning: Enabling generalized agent abilities for llms. *arXiv preprint arXiv:2310.12823*, 2023.

Di Zhang, Xiaoshui Huang, Dongzhan Zhou, Yuqiang Li, and Wanli Ouyang. Accessing gpt-4 level mathematical olympiad solutions via monte carlo tree self-refine with llama-3 8b. *arXiv preprint arXiv:2406.07394*, 2024.

Di Zhang, Jianbo Wu, Jingdi Lei, Tong Che, Jiatong Li, Tong Xie, Xiaoshui Huang, Shufei Zhang, Marco Pavone, Yuqiang Li, et al. Llama-berry: Pairwise optimization for olympiad-level mathematical reasoning via o1-like monte carlo tree search. In *NAACL*, 2025.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Livia Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. In *NeurIPS*, 2024.

Huichi Zhou, Yihang Chen, Siyuan Guo, Xue Yan, Kin Hei Lee, Zihan Wang, Ka Yiu Lee, Guchun Zhang, Kun Shao, Linyi Yang, et al. Agentfly: Fine-tuning llm agents without fine-tuning llms. *arXiv preprint arXiv:2508.16153*, 2025.

APPENDIX

A Ethic Statement

The study does not involve human subjects, data set releases, potentially harmful insights, applications, conflicts of interest, sponsorship, discrimination, bias, fairness concerns, privacy or security issues, legal compliance issues, or research integrity issues.

B Impact Statement

This work advances trustworthy foundation-model reasoning by introducing AlphaApollo. This system addresses two key failure modes: long-horizon multi-step solving and unreliable test-time refinement without trustworthy verification. AlphaApollo structures model-environment interaction through tool-integrated multi-turn reasoning, improves tool-use performance via turn-level reinforcement learning, and strengthens test-time reliability through a propose-judge-update evolution loop with tool-assisted verification and long-horizon memory. We expect these components to make LLM reasoning more reliable in practice, supporting safer deployment in settings where correctness matters. We do not anticipate significant negative societal impacts beyond general risks common to more capable reasoning systems.

C LLM Usage Disclosure

This submission was prepared with the assistance of LLMs, which were utilized for polishing content and checking grammar. The authors assume full responsibility for the entire content of the manuscript. It is confirmed that no LLM is listed as an author.

D Related Work

In this section, we systematically review prior work related to our three key features: tool-integrated reasoning in Sec. D.1, multi-model reasoning in Sec. D.2, and test-time iteration in Sec. D.3. Analogous to the Apollo Program, where *diverse experts* built *specialized tools* to *iteratively* launch the Apollo missions, these three features are essential to our AlphaApollo system.

D.1 Tool-integrated Reasoning

As aforementioned, the capabilities of FMs in tackling complex problems are limited by their insufficient computational ability and domain knowledge. Tool-integrated reasoning shows effectiveness in mitigating these shortcomings, enabling FMs to leverage external tools to bridge gaps in knowledge and arithmetic. The following introduces three directions for this paradigm.

Tool-integrated methods. Early tool-integrated methods leverage external tools to enhance FMs reasoning, addressing limitations in knowledge and computation. (1) For knowledge, Retrieval-Augmented Generation (RAG) (Lewis et al., 2020; Nakano et al., 2021) integrates knowledge bases to furnish FMs with essential facts for accurate inference, though its efficacy hinges on precise retrieval of contextually relevant information without noise (Gao et al., 2023b). To ensure retrieval reliability, Self-Ask (Press et al., 2022) decomposes queries into sub-questions for targeted retrieval, Self-RAG (Asai et al., 2024) verifies and filters irrelevant chunks, and GraphRAG (Edge et al., 2024) structures knowledge as context graphs to capture relational relevance. (2) For computation, Python has emerged as an effective tool for precise computation. Program-aided Language Models (PAL) (Gao et al., 2023a) and Program of Thoughts (PoT) (Chen et al., 2023b) incorporate Python code generation and execution within the reasoning process for arithmetic and logical tasks, while ViperGPT (Surís et al., 2023) extends this to vision, using code to process images and enrich multimodal reasoning.

Agentic frameworks. Beyond tool-integrated methods, agentic frameworks build flexible, tool-integrated environments that allow FMs to invoke tools dynamically, rather than adhering to predefined calling stages. For instance, SciMaster (Chai et al., 2025b) introduces agentic workflows that support dynamic reasoning via a Python-based tool for computation and retrieval, augmented by test-time scaling through reflection, solution refinement, and answer selection mechanisms to enhance FMs' ability on complex problems. Similarly, OctoTools (Lu et al., 2025) offers a reliable framework that integrates diverse tools through detailed tool cards describing their functions and utilities; it deploys a query analyzer agent to select a task-specific tool subset based on tool cards. This framework enables efficient tool usage when extensive tools are available, yielding more adaptive tool utilization for complex problems. Additionally, Alita (Qiu et al., 2025) proposes a framework for dynamically generating task-specific tools from code. It also leverages web search to iteratively refine both the reasoning process and the design of tools to optimize the solution for real-world tasks.

Learning frameworks. While the integration of multiple tools unlocks the reasoning potential of FMs, how to let FMs harness the usage of these tools in solving complex problems remains a significant challenge. To address this, tool-learning frameworks (Qin et al., 2024; Liu et al., 2024; Gao et al., 2024) enhance FMs' tool utilization through targeted post-training. Notably, several frameworks are tailored for tool-integrated long-horizon reasoning, enabling trainable multi-round interactions with tools. For instance, VerlTool (Jiang et al., 2025), RL-Factory (Chai et al., 2025a), and rStar2-Agent (Shang et al., 2025) employ unified tool managers to create model-friendly tool-use environments, supported by established RL training pipelines via VeRL (Sheng et al., 2024). Although these frameworks effectively integrate RL methods into tool-integrated reasoning, the inherent dynamics of such reasoning—particularly long-horizon planning and the incorporation of external tools—pose significant challenges to stable and efficient optimization. To mitigate this, Verl-Agent (Feng et al., 2025) introduces a step-independent rollout mechanism and customizable memory modules, alleviating inherent long-horizon reasoning difficulties. Whereas SimpleTIR (Xue et al., 2025) detects and filters trajectories featuring "void turns"—instances where reasoning collapses and destabilizes multi-turn agentic training—thereby promoting more robust optimization.

D.2 Multi-model Reasoning

Multi-model reasoning leverages the strengths of multiple models and allocates sub-tasks across models, which may adopt diverse roles rather than strictly complementary capabilities, to increase accuracy, robustness, and scalability in complex problem-solving. Representative paradigms include collaborative strategies and adversarial debate, with multi-agent fine-tuning further strengthening the system. We elaborate on each paradigm as follows.

Collaboration. Collaboration coordinates multiple models that work *synergistically*, with each contributing specialized capabilities toward a shared objective. AutoGen (Wu et al., 2024) provides a framework for multi-agent conversations that power next-generation FM applications, enabling agents to jointly tackle tasks such as coding and question answering. MetaGPT (Hong et al., 2024) employs role-based multi-agent collaboration and emphasizes structured, human-like workflows with standard operating procedures (SOPs), assigning roles such as product manager and engineer to inject domain expertise and improve efficiency in software-development tasks. In addition, HuggingGPT (Shen et al., 2023) leverages FMs to orchestrate Hugging Face models across modalities, integrating vision and speech to handle multimodal tasks.

Debate. Debate mechanisms engage multiple models in *mutual critique and refinement*, often paired with tool use to verify facts and resolve ambiguities. The MAD framework (Du et al., 2024) formalizes multi-round proposal, cross-examination, and revision among independent FMs before a final judgment, improving mathematical and strategic reasoning, reducing hallucinations, and working even with black-box models under task-agnostic prompts. CoA (Li et al., 2024) advances this paradigm with a sparse communication topology that lowers computational cost relative to fully connected settings while preserving reasoning performance in experiments with GPT and Mistral. Complementing these efforts, LLM-Coordination (Agashe et al., 2023) examines multi-agent behavior in pure coordination games, finding that FMs excel at environment comprehension yet underperform on theory-of-mind reasoning.

Multi-agent fine-tuning. Multi-agent fine-tuning jointly optimizes role-specialized models so that, as a group, they plan, call tools, and summarize more effectively than a single FM. Early systems fine-tune on agent trajectories (FireAct (Chen et al., 2023a)) or curated agent-instruction data (AgentTuning (Zeng et al., 2023)) to endow general agent abilities, providing a foundation for role-based optimization. An emerging direction explicitly fine-tunes a society of models from a common base using inter-agent data to diversify skills and improve coordination (Subramaniam et al., 2025). Shen et al. (2024) decomposes tool-learning into planner, caller, and summarizer roles, each fine-tuned on sub-tasks, achieving superior performance on ToolBench and surpassing single-FM approaches. AgentFly (Zhou et al., 2025) advances this paradigm by introducing memory-based online reinforcement learning for agent fine-tuning without updating FM weights. These methods demonstrate how multi-agent fine-tuning enhances coordination and tool usage in multi-turn tasks.

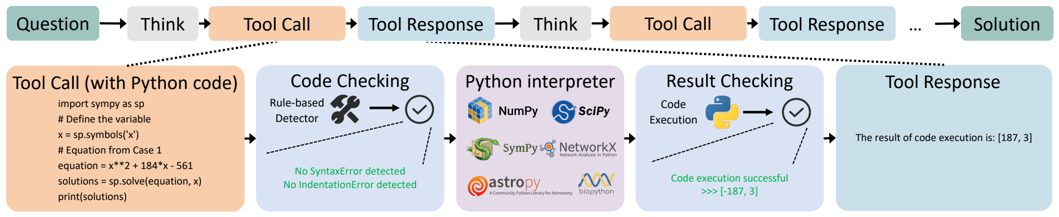
D.3 Test-time Iteration

Test-time iteration strategies improve FM reasoning by using extra computational resources to refine solutions or sample and verify diverse answers during inference, leveraging pretraining knowledge. Broadly, test-time iteration methods are categorized into parallel, sequential, and mixed strategies, distinguished by the interaction between iterations.

Parallel iteration. Most early test-time scaling methods utilize parallel iteration, where the model generates multiple independent reasoning trajectories. The final answer is then selected through various aggregation mechanisms. Self-Consistency (Wang et al., 2022) employs simple voting, Semantic Self-Consistency (Knappe et al., 2024) leverages semantic similarity, CISC (Taubenfeld et al., 2025) utilizes confidence metrics, and MCR (Yoran et al., 2023) implements model-decided final answers. Additionally, DIVSE (Naik et al., 2023) enhances the diversity of the reasoning trajectory by reformulating the original question to encourage exploration of a broader solution space. While this strategy enhances FM reasoning by allocating more computational budget to the reasoning process (Snell et al., 2024), it is fundamentally limited by the lack of interaction among different reasoning trajectories. This isolation limits the model in leveraging from previous attempts or refining its approach based on earlier outputs. The independence between trajectories constrains the potential for iterative improvement and fails to capitalize on insights that could emerge from comparing or combining intermediate reasoning steps (Qi et al., 2025).

Sequential iteration. Sequential iteration strategies leverage prior reasoning processes and outcomes to identify limitations and iteratively refine subsequent generations. Notably, Self-Refine methods (Madaan et al., 2023; Qu et al., 2024; Chen et al., 2023c) prompt FMs to revise initial outputs, generating improved solutions. Similarly, Muennighoff et al. (2025) employs a deliberation mechanism, replacing the end-of-sequence token with the keyword ‘wait’ to enable continuous reasoning post-answer generation. OPRO (Yang et al., 2024b) iteratively optimizes prompts based on prior outcomes to yield superior solutions. While these approaches enhance reasoning, their reliance on self-correction without external supervision can lead to unreliable outcomes (Huang et al., 2024). In contrast, Reflexion (Shinn et al., 2023), RCI (Kim et al., 2023), and Refiner (Paul et al., 2023) incorporate external feedback from environments, code executors, and critic models, respectively, providing robust supervision and improving performance. Furthermore, methods like Alpha-Evolve (Novikov et al., 2025) and TextGrad (Yuksekgonul et al., 2025) extend sequential iteration to scientific domains, demonstrating FMs’ potential in tackling complex challenges, such as code optimization and molecular synthesis.

Mixed iteration. Mixed iteration strategies integrate parallel and sequential iteration to combine the exploratory breadth of independent trajectories with the refinement depth of conditioned generations. This hybrid approach optimizes test-time compute allocation, often surpassing pure parallel or sequential methods on complex reasoning tasks by balancing exploration (through parallel iteration) and exploitation (via sequential refinement). A prominent example is Monte Carlo Tree Search (MCTS) (Silver et al., 2016; 2017), which employs multi-round node expansion through parallel iteration and outcome simulation via sequential iteration. Recent advancements apply MCTS to enhance FM reasoning by treating intermediate thoughts as tree nodes and reasoning outcomes as leaf nodes, iterating through node expansion and outcome simulation to derive optimized solutions (Zhang et al., 2024; Rabby et al., 2024; Zhang et al., 2025). Notably, Chang et al. (2025) incorporates self-refinement into node exploration, enabling MCTS to both identify the best node for subsequent iterations and optimize it for maximum exploitation.



```
import sympy as sp
# Define the variable
x = sp.symbols('x')
# Equation from Case 1
equation = x**2 + 184*x - 561
solutions = sp.solve(equation, x)
print(solutions)
```

```
>>> [-187, 3]
```

```
The result of code execution is: [187, 3]
```

Figure 10: The pipeline of the computational module in processing code.

E AGENTIC REASONING: IMPLEMENTATION DETAILS AND DISCUSSION

E.1 THE COMPUTATIONAL MODULE

This module integrates a Python interpreter with several external libraries of Python, as illustrated in Figure 10. Compared to other programming languages, Python's extensive ecosystem provides a more powerful computational environment for addressing complex reasoning tasks. Representative libraries are listed below:

- **SymPy** (Meurer et al., 2017) is a computer algebra system for symbolic mathematics, enabling the manipulation and solution of mathematical expressions in closed form. For example, it can determine the exact roots of the cubic equation $x^3 - 2x + 1$ within Python.
- **NumPy** (Harris et al., 2020) is a fundamental library for fast, vectorized numerical computing, providing powerful support for N-dimensional arrays and matrices. Typical tasks include linear algebra operations such as computing the dot product of two matrices.
- **SciPy** (Virtanen et al., 2020) is a comprehensive library of advanced numerical algorithms for science and engineering, supporting integration, ordinary differential equations, optimization, signal processing, sparse linear algebra, statistics, and more. For instance, it can readily solve problems like finding the minimum of $f(x) = \sin(x) + x^2$ over the interval $[-3, 3]$.

Python environment and code execution. Notably, this computational module shares the same Python environment as the AlphaApollo project, enabling users to easily extend the toolset by installing new Python libraries. When this module receives a tool-call request containing Python code from the model, it generates a temporary Python file and executes it in an individual subprocess, which is isolated from AlphaApollo's main process for safety reasons.

Error correction. The model-generated code can contain errors. To improve the robustness against these errors, AlphaApollo's computational module incorporates a rule-based error-correction. When the model generates Python code, the rule-based approach detects and automatically corrects errors identifiable through predefined rules, and then passes the corrected code to the Python interpreter for execution.

In rule-based error-correction, all `IndentationError` can be detected, with most being automatically correctable. The rule-based approach verifies the legality of indentation line by line, identifying and removing unnecessary space tokens to ensure proper indentation throughout the code. Additionally, certain `SyntaxError`, such as extraneous markdown blocks wrapping the code, can be detected and resolved. This method extracts the core Python code from wrapped content, ensuring the generated code is executable.

In addition, for errors such as `ValueError`, `TypeError`, `AttributeError`, and `ImportError`, which commonly arise when using external libraries (e.g., calling `sympy.factorial` with a negative argument, although it only accepts non-negative values), the computational module can invoke the retrieval module (Sec. E.2) for further guidance. By combining this capability with the two-fold error-correction mechanism, the computational module creates a model-friendly Python environment that empowers foundation models to perform code-augmented reasoning.

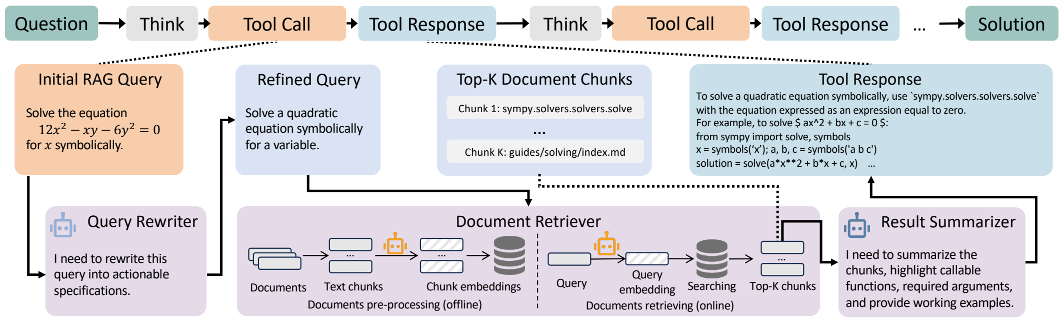

Question
Think
Tool Call
Tool Response
Think
Tool Call
Tool Response
...
Solution
Initial RAG Query
Solve the equation $12x^2 - xy - 6y^2 = 0$ for x symbolically.
Refined Query
Solve a quadratic equation symbolically for a variable.
Top-K Document Chunks
Chunk 1: sympy.solvers.solvers.solve
...
Chunk K: guides/solving/index.md
Tool Response
To solve a quadratic equation symbolically, use `sympy.solvers.solvers.solve` with the equation expressed as an expression equal to zero.
For example, to solve $ ax^2 + bx + c = 0 $:
from sympy import solve, symbols
x = symbols('x'); a, b, c = symbols('a b c')
solution = solve(a*x**2 + b*x + c, x) ...
Query Rewriter
I need to rewrite this query into actionable specifications.
Document Retriever
Documents
Text chunks
Chunk embeddings
Documents pre-processing (offline)
Query
Query embedding
Searching
Top-K chunks
Documents retrieving (online)
Result Summarizer
I need to summarize the chunks, highlight callable functions, required arguments, and provide working examples.

Figure 11: Schematic illustrating the information retrieval process of the retrieve module. The purple components correspond to the Retrieval Server. For clarity, Client 2 is omitted from the illustration.

E.2 The Retrieval Module

Foundation models demonstrate notable proficiency in generating code for widely used Python libraries. However, they often exhibit limitations when interfacing with less common libraries in solving complex problems, such as *NetworkX* for graphical problems or *SymPy* for symbolic mathematical problems. This can result in hallucinated or erroneous function calls, which compromise the reliability of tool integration. To mitigate this limitation, we augment AlphaApollo with a retrieval module that improves function invocation accuracy by *retrieving relevant functions and usage examples from library documentation*. Here, we offload the underlying complexities of information retrieval and processing to the retrieval module, allowing the (main) model to focus more on formulating the retrieval queries rather than selecting retrieval systems or information sources. As shown in Figure 11, the retrieval module comprises three core components: a query rewriter, a document retriever, and a result summarizer. The module follows a single-pass workflow that rewrites queries for clarity, retrieves semantically relevant documents, and summarizes documents to remove potentially redundant content. Specifically, each component functions as follows:

- **Query rewriter.** The workflow begins with the *query rewriter*, which transforms the initial query into a retrieval-friendly specification. Instead of passing detailed task descriptions directly, the rewriter abstracts them into generalizable forms that emphasize functional intent. For example, an over-detailed query "Solve the equation $12x^2 - xy - 6y^2 = 0$ for x symbolically" is rewritten as a more suitable one, "Solve a quadratic equation symbolically for a variable." This reformulation abstracts away numerical details while retaining the core intent, resulting in more relevant retrieval terms for the retriever. We implement the query rewriter using an instruction-following model with a tailored prompt template.

- **Document retriever.** After rewriting the query, the *document retriever* searches for relevant information in the module's indexed corpus, which includes the source code of a Python library and its associated documentation. Technically, we partition the corpus into *overlapped chunks*, which could improve retrieval effectiveness by aligning units with sentence-level semantics and reducing the inclusion of irrelevant content. We implement overlapped chunking with a fixed-length sliding window, allowing adjacent chunks to share overlapping tokens and thus preserve cross-boundary context. These chunks are then encoded by the embedding model and stored in a *vector database*. When the retriever receives a refined query from the rewriter, it first encodes the query using the same embedding model employed during document indexing. The retriever then conducts a cosine-similarity search over the database to locate the top-K chunk embeddings relevant to the query. The corresponding document segments are retrieved and assembled into a composite context, which is then passed to the summarization module.

- **Result summarizer.** Finally, the *result summarizer* filters and summarizes the retrieved context into a concise response. Its role is to highlight callable functions, required arguments, and minimal working examples rather than returning raw documentation. For instance, when the retriever provides (i) demonstrations of `sympy.solve`, (ii) module descriptions of `solveset`, and (iii) general API notes on `sympy`, the summarizer distills them into an actionable tool description. It identifies `solve` as the appropriate function, specifies its key arguments, and generates

System prompt of the first turn

You are a math problem solver agent tasked with solving the given math problem step-by-step.

Your question: {question}

Now it's your turn to respond to the current step.
You should first conduct the reasoning process. This process MUST be enclosed within <think ></think >tags.
After completing your reasoning, choose only one of the following actions (do not perform multiple actions at the same time):
1) <python_code>...</python_code>: If computation/checking is helpful, emit exactly ONE <python_code>...</python_code>block with pure Python 3. Inspect the <tool_response>(stdout from your code). If it disagrees with your reasoning, correct yourself.
2) <local_rag>...</local_rag>: You have access to a RAG System tool to search for documentation or examples (Supported repos: sympy, scipy, numpy, math, cmath, fractions, itertools). Emit exactly ONE <local_rag>...</local_rag>block with a JSON object. Inspect the returned <tool _response>(RAG result). If it disagrees with your reasoning, correct yourself. For example: <local_rag>{{"repo_name": "sympy", "query": "your query here"}}</local_rag>.
3) <answer>...</answer>: If you are ready to provide the self-contained solution, provide the answer only inside <answer>...</answer>, formatted in LaTeX, e.g., \\boxed{{...}}.

Figure 12: In the system prompt of the first turn, we instruct the model to solve math problems step by step, decide which tools to use, and present the final answer in a boxed format.

a compact example such as: `from sympy import solve, symbols; a, b, c, x = symbols('a b c x'); solve(ax**2 + bx + c, x).`

In summary, the retrieval module aims to improve function invocation in Python. It enables models to utilize external Python libraries to guide the generation and refinement of the generated code, grounding the model-generated code in reliable documentation.

E.3 Experimental Settings

In this setting, we enable tool-augmented reasoning as a baseline without any training or agentic evolution. The model performs multi-turn interactions with access to computational (Python code execution) and retrieval (local RAG) tools, limited to a maximum of 4 rounds ($T_{\text{max}} = 4$) to facilitate step-by-step problem-solving. We set the temperature to 0.6 for balanced exploration, with history length capped at 8 to maintain context across turns. No additional learning or verification mechanisms are applied, and each question is evaluated in a single-pass manner.

E.4 Prompts

In this part, we provide the employed prompts in the AlphaApollo in Figures 12 and 13.

F Agentic Learning: Implementation Details and Discussion

Prior agentic learning systems, such as VerlTool (Jiang et al., 2025) and RL-Factory (Chai et al., 2025a), stabilize agentic learning by employing a masking strategy that assigns zero weight to tokens from tool responses, thereby excluding them from optimization. Formally, considering the trajectory

System prompt for subsequent turns

You are a math problem solver agent tasked with solving the given math problem step-by-step.

Your question: {question}

Prior to this step, you have already taken {step_count} step(s). Below is the interaction history: {memory_context}
Now it's your turn to respond to the current step. You should first conduct the reasoning process. This process MUST be enclosed within <think> </think> tags.
After completing your reasoning, choose only one of the following actions (do not perform multiple actions at the same time):
1) <python_code>...</python_code>: If computation/checking is helpful, emit exactly ONE <python_code>...</python_code>block with pure Python 3. Inspect the <tool_response> (stdout from your code). If it disagrees with your reasoning, correct yourself.
2) <local_rag>...</local_rag>: You have access to a RAG System tool to search for documentation or examples (Supported repos: sympy, scipy, numpy, math, cmath, fractions, itertools). Emit exactly ONE <local_rag>...</local_rag>block with a JSON object. Inspect the returned <tool_response> (RAG result). If it disagrees with your reasoning, correct yourself. For example: <local_rag>{{"repo_name": "sympy", "query": "your query here"}}</local_rag>.
3) <answer>...</answer>: If you are ready to provide the self-contained solution, provide the answer only inside <answer>...</answer>, formatted in LaTeX, e.g., \\boxed{{...}}.

Figure 13: In the system prompt of subsequent turns, we integrate the memory from previous turns and instruct the model to continue solving the math problem step by step, decide which tools to use, and present the final answer in a boxed format.

sampling process of GRPO as an example, the objective for trajectory-level RL optimization is:

$$\mathcal{J}_{\text{Trajectory-GRPO}}(\theta) = \mathbb{E}_{\substack{p_1\sim\mathcal{D}\\ \{y_k^{(i)}\}_{i=1}^{N}\sim\pi_{\text{old}}(\cdot|p_1)}} \Bigg[\frac{1}{N}\sum_{i=1}^{N}\frac{1}{|y^{(i)}|}\sum_{k=1}^{|y^{(i)}|}\min\Bigg(m(y_k^{(i)})\frac{\pi_\theta\big(y_k^{(i)}\mid p_1, y_{<k}^{(i)}\big)}{\pi_{\text{old}}\big(y_k^{(i)}\mid p_1, y_{<k}^{(i)}\big)}A_k^{(i)},$$
$$\text{clip}\Bigg(m(y_t^{(i)})\frac{\pi_\theta\big(y_k^{(i)}\mid p_1, y_{<k}^{(i)}\big)}{\pi_{\text{old}}\big(y_k^{(i)}\mid p_1, y_{<k}^{(i)}\big)}, 1-\epsilon, 1+\epsilon\Bigg)A_k^{(i)}\Bigg) - \beta\mathbb{D}_{\text{KL}}\big[\pi_\theta\|\pi_{\text{ref}}\big]\Bigg] \tag{3}$$

where $y_k^{(i)}$ denotes k-th token of the (i)-th sampled trajectory, $m(y_k^{(i)}) = \mathbb{1}\big(y_k^{(i)} \in \mathcal{O}\big)$ indicates the masking strategy that optimize the policy with the model-generation content $\mathcal{O}$.

Similarly, the SFT in trajectory-level optimization learns given the full agentic reasoning trajectory while masking tokens from external tools.

$$\mathcal{J}_{\text{Trajectory-SFT}}(\theta) = \mathbb{E}_{(x,y)\sim\mathcal{D}}\left[\sum_{k=1}^{|y|} m(y_k)\log\pi_\theta(y_k \mid y_{<k}, x)\right], \tag{4}$$

Although the masking strategy prevents the policy from learning tool responses, thereby enhancing stability, the long-horizon nature of agentic reasoning can still destabilize training. Low-probability tokens arising in extended trajectories may compromise optimization, leading to reward collapse, as evidenced in studies on agentic reinforcement learning (Xue et al., 2025; Wang et al., 2025b).

F.1 Experimental Settings

In this phase, we apply a learning rate of 1e-6, a training batch size of 128, and a group size of 8. The temperature is set to 0.4 to ensure stable performance monitoring throughout the reinforcement learning process. The training datasets consist of Lighteval-Math, LIMR, and DeepScaleR for comprehensive mathematical reasoning coverage. To demonstrate the generality of our framework, we trained on each of these datasets separately in our experiments.

F.2 Prompts

We use the same prompts as in Appendix E.4, shown in Figures 12 and 13.

G Agentic Evolution: Implementation Details and Discussion

G.1 Experimental Settings

Table 4: The detailed settings of models in agentic evolution. Here, output length refers to the maximum response length specified during evaluation of evolution.

Model Family	**Model Name**	Maximum context length	Output length
Qwen2.5 (Yang et al., 2024a)	Qwen2.5-1.5B-Instruct	128K	8k
	Qwen2.5-3B-Instruct	128K	8k
	Qwen2.5-7B-Instruct	128K	8k
	Qwen2.5-14B-Instruct	128K	8k

We show the configuration of agentic evolution in Table 4. This phase involves no training and utilizes a dual-model configuration. The policy model operates with a temperature of 0.7 to encourage exploration and diverse solution generation, while the verifier uses a temperature of 0.4. The self-evolving process runs for 10 rounds, the verifier conducts repeated verification for 5 times to ensure majority judgment. Both models share a maximum token limit of 8k to accommodate complex reasoning trajectories.

G.2 Prompts

In this part, we present the detailed system prompts governing the specialized agents within our agentic evolution framework (Sec. 2.3). These prompts dictate the specific behaviors, tool permissions, and interaction protocols for each role.

`Solver`: The Solver is responsible for generating solution trajectories. In the initial round, the prompts follow the standard protocols shown in Figures 12 and 13 (Sec. E.4). In subsequent rounds, the solver is augmented with long-term memory entities. Figure 14 illustrates the initialization prompt containing the history of past attempts ($\mathcal{M}^{(r)}$), while Figure 15 shows the instruction for intermediate steps, which integrates both the previous attempts and the local interaction context.

`Abstractor`: To facilitate efficient memory storage, the Abstractor (Figure 16) is tasked with compressing raw interaction logs. It filters out computational noise, syntax errors, and backtracking, restructuring the successful reasoning path into concise “logical checkpoints”. This process distinguishes between computed values and intuitive assertions, ensuring the stored memory is high-quality and retrieval-ready.

`Evaluator`: As shown in Figure 17, the agent is instructed to verify the solver’s candidate solution against the original problem. The prompt enforces a ”Think-Check-Report” lifecycle, explicitly authorizing the use of tools to validate intermediate steps and definitions before issuing a binary correctness judgment. For multi-turn verification sessions, the prompt in Figure 18 adapts to include the interaction history.

`Summarizer`: Designed for ensemble verification scenarios, the Summarizer (Figure 19) is activated when multiple evaluators reach a consensus. It aggregates individual verification into a single judgement by synthesizing the arguments and removing redundancy, thereby providing a comprehensive justification.

Prompt for `Solver` with $\mathcal{M}^{(r)}$ for the first turn

You are a math problem solver agent tasked with solving the given math problem step-by-step.

Your question: {question}

Below are the previous solutions and their verification feedback: {previous_solutions}

The \boxed{1} within feedback indicates that the previous solution was correct, and \boxed{0} indicates that the previous solution was incorrect. Use the previous solutions and their verification feedback to guide your current step. Remember that if the previous solution was incorrect, you should correct your reasoning and try again.

Now it's your turn to respond to the current step. You should first conduct the reasoning process. This process MUST be enclosed within <think> </think> tags. After completing your reasoning, choose only one of the following actions (do not perform multiple actions at the same time):

1) <python_code>...</python_code>: If computation/checking is helpful, emit exactly ONE <python_code>...</python_code>block with pure Python 3. Inspect the <tool_response> (stdout from your code). If it disagrees with your reasoning, correct yourself.
2) <local_rag>...</local_rag>: You have access to a RAG System tool to search for documentation or examples (Supported repos: sympy, scipy, numpy, math, cmath, fractions, itertools). Emit exactly ONE <local_rag>...</local_rag>block with a JSON object. Inspect the returned <tool_response> (RAG result). If it disagrees with your reasoning, correct yourself. For example: <local_rag>{{"repo_name": "sympy", "query": "your query here"}}</local_rag>.
3) <answer>...</answer>: If you are ready to provide the self-contained solution, provide the answer only inside <answer>...</answer>, formatted in LaTeX, e.g., \\boxed{...}.

Figure 14: Prompt for `Solver` with $\mathcal{M}^{(r)}$ for the first turn. The system instruction for the first turn, incorporating the set of previous solutions and their validity ($\mathcal{M}^{(r)}$).

H Case Studies

H.1 Python Code Errors

In this section, we show cases of model-generated Python code scripts with the AlphaApollo framework. Therein, partial `SyntaxError` and `IndentationError` can be solved with the rule-based error correction component as mentioned in Sec. E.1, while other errors that cannot be solved with rules are refined with the model-based error correction component. We list them as follows:

- Solved with the rule-based error correction component.
  - `SyntaxError`(Figure 20);
  - `IndentationError` (Figure 21);
- Solved with the model-based error correction component.
  - `SyntaxError` (Figure 22);
  - `NameError` (Figure 23);
  - `IndexError` (Figure 24);
  - `TypeError` (Figure 25);
  - `ValueError` (Figure 26);
  - `ImportError` (Figure 27);
  - `AttributeError` (Figure 28);
  - `NotImplementedError` (Figure 29).

Prompt for `Solver` with $\mathcal{M}^{(r)}$ for subsequence turn

You are a math problem solver agent tasked with solving the given math problem step-by-step.

Your question: {question}

Below are the previous solutions and their verification feedback: {previous_solutions}

The \boxed{1} within feedback indicates that the previous solution was correct, and \boxed{0} indicates that the previous solution was incorrect. Use the previous solutions and their verification feedback to guide your current step. Remember that if the previous solution was incorrect, you should correct your reasoning and try again.

Prior to this step, you have already taken {step_count} step(s). Below is the interaction history: {memory_context}

Now it's your turn to respond to the current step. You should first conduct the reasoning process. This process MUST be enclosed within <think> </think> tags. After completing your reasoning, choose only one of the following actions (do not perform multiple actions at the same time):

1) <python_code>...</python_code>: If computation/checking is helpful, emit exactly ONE <python_code>...</python_code>block with pure Python 3. Inspect the <tool_response> (stdout from your code). If it disagrees with your reasoning, correct yourself.
2) <local_rag>...</local_rag>: You have access to a RAG System tool to search for documentation or examples (Supported repos: sympy, scipy, numpy, math, cmath, fractions, itertools). Emit exactly ONE <local_rag>...</local_rag>block with a JSON object. Inspect the returned <tool_response> (RAG result). If it disagrees with your reasoning, correct yourself. For example: <local_rag>{{"repo_name": "sympy", "query": "your query here"}}</local_rag>.
3) <answer>...</answer>: If you are ready to provide the self-contained solution, provide the answer only inside <answer>...</answer>, formatted in LaTeX, e.g., \\boxed{...}.

Figure 15: Prompt for `Solver` with $\mathcal{M}^{(r)}$ for the subsequent turns. Instructions for subsequent turns where the agent must integrate the interaction history (`memory_context`) with the evolutionary feedback from $\mathcal{M}^{(r)}$ to advance the solution step-by-step.

Prompt for `Abstractor`

ROLE
You are a Mathematical Logic Auditor. Compress the interaction log into a Verification Brief.

INPUT DATA
The log contains '<think>', '<python_code>', '<tool_response>', and '<answer>' tags.

PROTOCOL
1. Filter Noise: Retain only the mathematical setup, derived constants, and successful logic. Discard syntax errors, backtracking, and internal monologue.
2. Track Origins: Explicitly differentiate between values computed via code and those asserted via intuition or external knowledge.
3. Format: Use LaTeX for math. Present as sequential Logical Checkpoints.

OUTPUT TEMPLATE
Strategy: [Brief summary of the approach]

Logical Checkpoints:
1. Setup: [Variable definitions and initial conditions]
2. Intermediate Result: [Key derived values]
3. Pivotal Step: [Crucial logic or calculation]
4. Resolution: [How the final result was reached]

Final Claim: <answer>...</answer>

TASK
Summarize:
{content}

Figure 16: System instructions for `Abstractor`. The agent generates a structured summary by extracting the key mathematical setup, intermediate results, and the pivotal logic leading to the final answer, while discarding irrelevant content.

Prompt for `Evaluator` for the first turns.

You are a math verifier agent whose only job is to check whether the solver agent's proposed solution is correct.

Original question:
{question}

Solver's latest solution attempt:
{solver_solution}

Now it's your turn to respond to the current step. You should first conduct the reasoning process. This process MUST be enclosed within <think></think>tags. After completing your reasoning, choose only one of the following actions (do not perform both):

1) <python_code>...</python_code>: If computation/checking is helpful, emit exactly ONE <python_code>...</python_code>block with pure Python 3. Inspect the <tool_response>(stdout from your code). If it disagrees with your reasoning, correct yourself.
2) <local_rag>...</local_rag>: You have access to a RAG System tool to search for documentation or examples (Supported repos: sympy, scipy, numpy, math, cmath, fractions, itertools). Emit exactly ONE <local_rag>...</local_rag>block with a JSON object. Inspect the returned <tool_response> (RAG result). If it disagrees with your reasoning, correct yourself. For example: <local_rag>{{"repo_name": "sympy", "query": "your query here"}}</local_rag>.
3) <report>...</report>: If you are ready to conclude, wrap your verification report inside <report>...</report>tags. The report should:
- Clearly state whether the policy solution appears correct or incorrect.
- Explain the key reasoning behind your judgment (keep it concise).
- End with your judgement in the format: \boxed{{1}} if correct, or \boxed{{0}} if incorrect.
- The judgement should be enclosed within <report>...</report>tags.

Figure 17: The system instructions for the evaluator agent for the first turn. It receives the problem and the candidate solution, utilizes available tools to audit the logic, and outputs a verification.

Prompt for `Evaluator` for the subsequence turns.

You are a math verifier agent whose only job is to check whether the solver agent's proposed solution is correct.

Original question:
{question}

Solver's latest solution attempt:
{solver_solution}

Prior to this step, you have already taken {step_count} step(s). Below is the interaction history: {memory_context}

Now it's your turn to respond to the current step. You should first conduct the reasoning process. This process MUST be enclosed within <think></think>tags. After completing your reasoning, choose only one of the following actions (do not perform both):

1) <python_code>...</python_code>: If computation/checking is helpful, emit exactly ONE <python_code>...</python_code>block with pure Python 3. Inspect the <tool_response>(stdout from your code). If it disagrees with your reasoning, correct yourself.
2) <local_rag>...</local_rag>: You have access to a RAG System tool to search for documentation or examples (Supported repos: sympy, scipy, numpy, math, cmath, fractions, itertools). Emit exactly ONE <local_rag>...</local_rag>block with a JSON object. Inspect the returned <tool_response> (RAG result). If it disagrees with your reasoning, correct yourself. For example: <local_rag>{{"repo_name": "sympy", "query": "your query here"}}</local_rag>.
3) <report>...</report>: If you are ready to conclude, wrap your verification report inside <report>...</report>tags. The report should:
- Clearly state whether the policy solution appears correct or incorrect.
- Explain the key reasoning behind your judgment (keep it concise).
- End with your judgement in the format: \boxed{{1}} if correct, or \boxed{{0}} if incorrect.
- The judgement should be enclosed within <report>...</report>tags.

Figure 18: The system instruction for `Evaluator` for subsequent turns.

Prompt for `Summarizer`

You are a math verification report aggregator. Multiple evaluators have independently verified a solver agent's solution and reached the same judgment. Your task is to synthesize their reports into a single, comprehensive report.

Original question:
{question}

Solver agent's solution:
{Solver_solution}

The evaluators have all concluded with judgment: \boxed{{{majority_judgment}}}

Below are the individual reports:
{individual_reports}

Your task:
1. Analyze the key reasoning points from each verifier report.
2. Synthesize the most compelling arguments and evidence into a single, coherent report.
3. Ensure the aggregated report is comprehensive yet concise.
4. Maintain the same final judgment as the individual reports.

Wrap your aggregated report inside <report>...</report>tags. The report should:
- Clearly state whether the policy solution is correct or incorrect.
- Combine the strongest reasoning from all verifier reports.
- End with the judgment: \boxed{majority_judgment}

Figure 19: Instructions for summarizing verifications. The agent summarizes the key information from several concordant evaluators into a single representative judgment, maintaining the original consensus judgment.

Python Code

```
““““python
number = 2024
binary_str = bin(number)[2:]
print(f“Binary representation of number: binary_str”)
positions = [i for i, bit in enumerate(reversed(binary_str)) if bit == ‘1’]
print(f“Positions with 1s (from right, starting at 0): positions”)
””””
```

Execution Result

```
File “main.py”, line 7
    ””””
    ^
SyntaxError: unterminated triple-quoted string literal (detected at line 7)
```

Figure 20: The example of `SyntaxError` that can be solved with rules. We observe model-generated code often contains markdown code block format (*i.e.*, ““““python ... ””””), which is invalid for the Python interpreter. To mitigate this kind of error, the rule-based error correction component automatically detects the format of code blocks and eliminates them before executing the code.

Python Code

```
import sympy as sp

y = sp.symbols('y')
   equation = y**3 - 27*y + 46
   roots = sp.solve(equation, y)
   print(roots)
```

Execution Result

```
File "main.py", line 4
        equation = y**3 - 27*y + 46
        ^
IndentationError: unexpected indent
```

Figure 21: The example of `IndentationError` in model-generated code scripts. The rule-based error correction component automatically identifies the unexpected indentation in the code script and fixes it.

Python Code

```
from itertools import combinations_with_replacement
# Calculate the number of non-negative integer solutions to a + b + c = 8
# Using stars and bars formula: C(n + k -1, k-1) where n=8, k=3
n = 8
k = 3
solutions = (n + k -1) choose (k-1)
# But let's compute it with code as well
count = 0
for a in range(0, 9):
    for b in range(0, 9 - a):
        c = 8 - a - b
        if c >= 0 and c <=9:
            count +=1
print(count)
```

Execution Result

```
File "main.py", line 6
        solutions = (n + k -1) choose (k-1)
                               ^^^^^^
SyntaxError: invalid syntax
```

Figure 22: The example of `SyntaxError` that cannot be solved with rules. The model-based error correction components provide specific feedback to the model via the tool response that instructs the model to identify and solve this error.

Python Code

```
from fractions import Fraction

# calculate combinations
c0_4 = math.comb(4, 4)
c1_4 = math.comb(4, 3)
c2_4 = math.comb(4, 2)
c3_4 = math.comb(4, 1)

# Compute probabilities using exact fractions
P_A = Fraction(c1_4, c0_4 + c1_4 + c2_4 + c3_4)
numerator_B = c0_4 + c1_4 * c6_3
P_B = Fraction((numerator_B), c0_4)

conditional_prob = P_A / P_B
```

Execution Result

```
Traceback (most recent call last):
    File "stdin", line 4, in
      c0_4 = math.comb(4, 4)
             ^^^^
NameError: name 'math' is not defined
```

Figure 23: The example of `NameError` in model-generated code scripts. The model-based error correction component provides the likely root reason of this error (*i.e.,* the model mistakenly uses undefined variables or Python packages) to help the model identify and fix this error.

Python Code

```
from sympy import symbols, solve, sqrt
# Define the variables
x, y, z = symbols('x y z')
# Solve for vertices under the condition x <y <z
# First vertex: x = y
x1 = 25
y1 = 25
z1 = 75 - 2*x1
# Second vertex: y = z
y2 = 25
x2 = 75 - 2*y2
z2 = y2
# Third vertex: x = z
z3 = 25
x3 = z3
y3 = 75 - 2*z3
# Ensure x <y <z
vertices = [(x1, y1, z1), (x2, y2, z2), (x3, y3, z3)]
vertices = [(x, y, z) for x, y, z in vertices if x <y <z]
# Calculate the area of the triangle
area = sqrt((vertices[0][0] - vertices[1][0])**2 + (vertices[0][1] - vertices[1][1])**2 \
    + (vertices[0][2] - vertices[1][2])**2) * sqrt((vertices[0][0] - vertices[2][0])**2 \
    + (vertices[0][1] - vertices[2][1])**2 + (vertices[0][2] - vertices[2][2])**2) \/ 2
# Simplify the area
area_simplified = area.simplify()
area_simplified
```

Execution Result

```
Traceback (most recent call last):
  File "main.py", line 21, in <module>
    area = sqrt((vertices[0][0] - vertices[1][0])**2 + (vertices[0][1] - vertices[1][1])**2 + (vertices[0][2] - vertices[1][2])**2) * sqrt((vertices[0][0] - vertices[2][0])**2 + (vertices[0][1] - vertices[2][1])**2 + (vertices[0][2] - vertices[2][2])**2) \/ 2
IndexError: list index out of range
```

Figure 24: The example of `IndexError` in model-generated code scripts.

Python Code

```
import numpy as np
from scipy.optimize import root_scalar
# Define the functions
def f(x):
  return abs(abs(x) - 1/2)
def g(x):
  return abs(abs(x) - 1/4)
def y_func(x):
  return 4 * g(f(np.sin(2 * np.pi * x)))
def x_func(y):
  return 4 * g(f(np.cos(3 * np.pi * y)))
def equation(x):
  y = y_func(x)
  x_prime = x_func(y)
  return x_prime - x
# Use root_scalar to find roots in [0, 1]
# Initial guess
initial_guess = 0.5
# Solve using bisection method
result = root_scalar(equation, bracket=[0, 1], method='bisection', tol=1e-6)
# Print the root
print(f"Root found at x = {result.root .6f}")
# Check for other roots by evaluating the function at different points
x_values = np.linspace(0, 1, 1000)
function_values = [equation(x) for x in x_values]
# Count the number of sign changes
sign_changes = 0
for i in range(len(function_values) - 1):
  if function_values[i] * function_values[i+1] < 0:
    sign_changes += 1
print(f"Number of sign changes (potential roots): {sign_changes}")
```

Execution Result

```
Traceback (most recent call last):   File "main.py", line 20, in <module>
    result = root_scalar(equation, bracket=[0, 1], method='bisection', tol=1e-6)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

TypeError: root_scalar() got an unexpected keyword argument 'tol'
```

Figure 25: The example of `TypeError` in model-generated code scripts. The reasoning model is instructed by the model-based error correction component to check the correctness of the type of variables.

Python Code

```
import numpy as np
# Define h(x)
def h(x):
    z = abs(np.sin(2 * np.pi * x))
    if z <= 0.5:
        f_z = 0.5 - z
    else:
        f_z = z - 0.5
    if f_z <= 0.25:
        g_f_z = 0.25 - f_z
    else:
        g_f_z = f_z - 0.25
    return 4 * g_f_z
# Define k(y)
def k(y):
    w = abs(np.cos(3 * np.pi * y))
    if w <= 0.5:
        f_w = 0.5 - w
    else:
        f_w = w - 0.5
    if f_w <= 0.25:
        g_f_w = 0.25 - f_w
    else:
        g_f_w = f_w - 0.25
    return 4 * g_f_w
# Define F(x) = x - k(h(x))
def F(x):
    return x - k(h(x))
# Evaluate F(x) over a grid of x values
x_values = np.linspace(0, 1, 100000)
F_values = F(x_values)
# Count the number of sign changes in F_values
sign_changes = 0
for i in range(len(F_values) - 1):
    if F_values[i] * F_values[i+1] <0:
        sign_changes += 1
# Print the number of sign changes
print(f'Number of sign changes: {sign_changes}')
```

Execution Result

```
Traceback (most recent call last):
    File "main.py", line 35, in <module>
        F_values = F(x_values)
                   ^^^^^^^^^^^^^^
    File "main.py", line 31, in F
        return x - k(h(x))
                     ^^^^
    File "main.py", line 6, in h
        if z <= 0.5:
           ^^^^^^^^^^^
ValueError: The truth value of an array with more than one element is ambiguous. Use
a.any() or a.all()
```

Figure 26: The example of `ValueError` in model-generated code scripts. The model-based error correction component instructs the reasoning model to verify the validity of variable values.

Python Code

```
from sympy import symbols, Eq, solve, maximize
x, y, z = symbols('x y z')
# Constraints
constraint1 = Eq(x*y + y*z + z*x, 27)
constraint2 = Eq(x*y*z, 23)
# Function to maximize
objective = x**2 + y**2 + z**2
# Use Lagrange multipliers
lambda1, lambda2 = symbols('lambda1 lambda2')
L = objective - lambda1*(constraint1.lhs - constraint1.rhs) \
    - lambda2*(constraint2.lhs - constraint2.rhs)
# Partial derivatives
partial_x = L.diff(x)
partial_y = L.diff(y)
partial_z = L.diff(z)
partial_lambda1 = L.diff(lambda1)
partial_lambda2 = L.diff(lambda2)
# Solve the system of equations
solutions = solve([partial_x, partial_y, partial_z, partial_lambda1, partial_lambda2], (x, y, z,
lambda1, lambda2))
solutions
```

Execution Result

```
Traceback (most recent call last):
    File "main.py", line 1, in <module>
        from sympy import symbols, Eq, solve, maximize
ImportError: cannot import name 'maximize' from 'sympy'
(/usr/local/lib/python3.11/dist-packages/sympy/__init__.py)
```

Figure 27: The example of `ImportError` in model-generated code scripts. The model-based error correction component provides instructions for the reasoning model to verify the imported packages in the code script.

Python Code

```
import numpy as np
from scipy.optimize import root_scalar
# Define the functions
def f(x):
    return abs(abs(x) - 1/2)
def g(x):
    return abs(abs(x) - 1/4)
def y_func(x):
    return 4 * g(f(np.sin(2 * np.pi * x)))
def x_func(y):
    return 4 * g(f(np.cos(3 * np.pi * y)))
def equation(x):
    y = y_func(x)
    x_prime = x_func(y)
    return x_prime - x
# Use root_scalar to find roots in [0, 1]
# Initial guess
initial_guess = 0.5
# Solve using bisection method
result = root_scalar(equation, bracket=[0, 1], method='bisection', rtol=1e-6)
# Print the root
print(f"Root found at x = {result.root .6f}")
# Check for other roots by evaluating the function at different points
x_values = np.linspace(0, 1, 1000)
function_values = [equation(x) for x in x_values]
# Count the number of sign changes
sign_changes = 0
for i in range(len(function_values) - 1):
    if function_values[i] * function_values[i+1] <0:
        sign_changes += 1
print(f"Number of sign changes (potential roots): {sign_changes}")
```

Execution Result

```
Traceback (most recent call last):
    File "/usr/local/lib/python3.11/dist-packages/scipy/optimize/_root_scalar.py", line 276,
in root_scalar
        methodc = getattr(optzeros, map2underlying.get(meth, meth))
                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

AttributeError: module 'scipy.optimize._zeros_py' has no attribute 'bisection'. Did you
mean: 'bisect'?
The above exception was the direct cause of the following exception:
Traceback (most recent call last):
    File "main.py", line 20, in <module>
        result = root_scalar(equation, bracket=[0, 1], method='bisection', rtol=1e-6)
                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

    File "/usr/local/lib/python3.11/dist-packages/scipy/optimize/_root_scalar.py", line 278,
in root_scalar
        raise ValueError(f'Unknown solver {meth}' ) from e
ValueError: Unknown solver bisection
```

Figure 28: The example of `AttributeError` in model-generated code scripts. The model-based error correction component gives feedback to the reasoning model to verify the called attribute of using variables.

Python Code

```
import sympy as sp

x, y = sp.symbols('x y')

# Equation of AB
ab_eq = sp.Eq(y, -sp.sqrt(3)x + sp.sqrt(3)/2)

# Equation of the envelope
envelope_eq = sp.Eq(x (2/3) + y (2/3), 1)

# Substitute y from ab_eq into envelope_eq
substituted_eq = envelope_eq.subs(y, ab_eq.rhs)

# Solve the equation
solution = sp.solve(substituted_eq, x)

solution
```

Execution Result

```
Traceback (most recent call last):
    File "main.py", line 15, in <module>
        solution = sp.solve(substituted_eq, x)
                   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

    File "/usr/local/lib/python3.11/dist-packages/sympy/solvers/solvers.py", line 1170, in
solve
        solution = _solve(f[0], symbols, flags)
                   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

    File "/usr/local/lib/python3.11/dist-packages/sympy/solvers/solvers.py", line 1729, in
_solve
        raise NotImplementedError('\n'.join([msg, not_impl_msg # fi]))
NotImplementedError: multiple generators {(-2*sqrt(3)*x + sqrt(3))**{(1/3)},
x**{(1/3)}} No algorithms are implemented to solve equation x**{2/3} + (-sqrt(3)*x +
sqrt(3)/2)**{2/3} - 1
```

Figure 29: The example of `NotImplementedError` in model-generated code scripts.

H.2 Cognitive Behaviors

In this section, we provide the compressed model responses in Figure 30 for quick reference and the complete responses in Figures 31, 32, 33, and 34 for the case studies in Sec. 3.

Case study: Decomposition

. . . , we need to **break it down into several steps**: . . .
Let's start by calculating . . .

Python Interpreter Calling

```
from math import comb
(. . . )
```

The result of the code execution is: 210

Next, let's calculate . . .

Python Interpreter Calling

```
from math import comb
(. . . )
```

The result of the code execution is: 115, 90, 24, 1
. . .
Thus, the final answer is: $\boxed{116}$.

Case study: Correction

. . . Let's write a Python script . . .

Python Interpreter Calling

```
def find_largest_N():
(. . . )
```

The result of the code execution is: None
It seems there was an issue . . . **Let's re-evaluate . . .**
. . .
We will re-run the script with a more detailed check . . .

Python Interpreter Calling

```
def find_largest_N():
(. . . )
```

The result of the code execution is: 5694
. . .
Thus, the final answer is: $\boxed{699}$.

Case study: Verification

. . . Now, we need to compute xy:

$$xy = \left(\frac{10}{a}\right)\left(\frac{5a}{2}\right) = \frac{50a}{2a} = 25$$

Let's verify the solution . . .

Python Interpreter Calling

```
from sympy import symbols, Eq, solve, log
x, y, a = symbols('x y a')
eq1 = Eq(x*a, 10)
eq2 = Eq(4*y/a, 10)
sol = solve((eq1, eq2), (x, y))
xy = sol[x] * sol[y]
xy.simplify()
```

The result of the code execution is: 25
The value of xy is $\boxed{25}$.

Case study: Backtracking

. . . Let's implement the above steps . . .

Python Interpreter Calling

```
from sympy import *
(. . . )
```

TypeError: cannot determine truth value of Relational.
It appears that the direct symbolic solution approach using SymPy did not work as expected due to the complexity of the equations.
Let's try a different approach . . .

Python Interpreter Calling

```
# Iterate over possible values of a and b
(. . . )
```

The result of the code execution is: 601
Thus, the final answer is: $\boxed{601}$.

Figure 30: Case studies of models' cognitive behaviors exhibited by AlphaApollo.

Case study: Decomposition

Problem

Jen enters a lottery by picking 4 distinct numbers from $S = \{1, 2, 3, \cdots, 9, 10\}$. 4 numbers are randomly chosen from S. She wins a prize if at least two of her numbers were 2 of the randomly chosen numbers, and wins the grand prize if all four of her numbers were the randomly chosen numbers. The probability of her winning the grand prize given that she won a prize is $\frac{m}{n}$ where m and n are relatively prime positive integers. Find $m + n$.

To solve this problem, we need to break it down into several steps:
1. **Determine the total number of ways to pick 4 numbers out of 10**: This is a combination problem where we choose 4 numbers from a set of 10. The number of ways to do this is denoted as $C(10, 4)$.
2. **Determine the number of favorable outcomes for winning a prize**: - **Winning a prize**: Jen's 4 chosen numbers must have at least 2 numbers in common with the 4 randomly chosen numbers. - **Winning the grand prize**: Jen's 4 chosen numbers must be exactly the same as the 4 randomly chosen numbers.
3. **Calculate the conditional probability**: - We need to find the probability of winning the grand prize given that she has won a prize.
Let's start by calculating the total number of ways to pick 4 numbers out of 10.

Python Interpreter Calling

```
from math import comb
# Total number of ways to pick 4 numbers out of 10
C_10_4 = comb(10, 4)
C_10_4
```

The result of the code execution is: 210
Next, let's calculate the number of favorable outcomes for winning a prize. This involves counting the number of ways Jen's 4 chosen numbers can have at least 2 numbers in common with the 4 randomly chosen numbers.We will break this down further:
- **Exactly 2 numbers in common**: We choose 2 numbers from the 4 picked by Jen and 2 numbers from the remaining 6 numbers.
- **Exactly 3 numbers in common**: We choose 3 numbers from the 4 picked by Jen and 1 number from the remaining 6 numbers.
- **Exactly 4 numbers in common**: We choose all 4 numbers from the 4 picked by Jen.
Let's compute these values.

Python Interpreter Calling

```
from math import comb
# Number of ways to have exactly 2 numbers in common
C_4_2 = comb(4, 2)
C_6_2 = comb(6, 2)
ways_2_common = C_4_2 * C_6_2

# Number of ways to have exactly 3 numbers in common
C_4_3 = comb(4, 3)
C_6_1 = comb(6, 1)
ways_3_common = C_4_3 * C_6_1

# Number of ways to have exactly 4 numbers in common
C_4_4 = comb(4, 4)
C_6_0 = comb(6, 0)
ways_4_common = C_4_4 * C_6_0

# Total number of favorable outcomes for winning a prize
favorable_winning_prize = ways_2_common + ways_3_common + ways_4_common

favorable_winning_prize, ways_2_common, ways_3_common, ways_4_common
```

The result of the code execution is: 115 90 24 1

The calculations yield the following results: - Number of ways to have exactly 2 numbers in common: 90 - Number of ways to have exactly 3 numbers in common: 24 - Number of ways to have exactly 4 numbers in common: 1
Thus, the total number of favorable outcomes for winning a prize is:

$$90 + 24 + 1 = 115$$

Now, we need to find the probability of winning the grand prize given that she has won a prize. This is the ratio of the number of ways to win the grand prize to the total number of ways to win a prize:

$$P(\text{Grand Prize} \mid \text{Prize}) = \frac{\text{Number of ways to win the grand prize}}{\text{Total number of favorable outcomes for winning a prize}} = \frac{1}{115}$$

Since 1 and 115 are relatively prime, the fraction is already in its simplest form. Therefore, $m = 1$ and $n = 115$, and the sum $m + n$ is:

$$1 + 115 = 116$$

Thus, the final answer is: $\boxed{116}$.

Figure 31: The complete model response for the Decomposition case study.

Case study: Correction

Problem

Let N be the greatest four-digit positive integer with the property that whenever one of its digits is changed to 1, the resulting number is divisible by 7. Let Q and R be the quotient and remainder, respectively, when N is divided by 1000. Find $Q + R$.

To find the greatest four-digit positive integer N with the property that changing any one of its digits to 1 results in a number divisible by 7, we need to follow these steps:
1. **Understand the Problem:**
- We need N such that $N \equiv 0 \pmod 7$.
- Changing any digit of N to 1 should also result in a number divisible by 7.
2. **Formulate the Constraints:**
- Let $N = 1000a + 100b + 10c + d$, where a, b, c, and d are the digits of N.
- For each digit $x \in \{a, b, c, d\}$, the number formed by changing x to 1 must be divisible by 7.
3. **Check the Conditions:**
- If a is changed to 1, the new number is $1000 + 100b + 10c + d$. This must be divisible by 7:

$$1000 + 100b + 10c + d \equiv 0 \pmod 7$$

- Similarly, for b, c, and d:

$$1000a + 100 + 10c + d \equiv 0 \pmod 7$$

$$1000a + 100b + 10 + d \equiv 0 \pmod 7$$
$$1000a + 100b + 10c + 1 \equiv 0 \pmod 7$$

4. **Simplify the Conditions:**
- Since $1000 \equiv 6 \pmod 7$, $100 \equiv 2 \pmod 7$, and $10 \equiv 3 \pmod 7$, we can rewrite the conditions as:

$$6a + 2b + 3c + d + 1 \equiv 0 \pmod 7$$
$$6a + 2b + 3c + d + 2 \equiv 0 \pmod 7$$
$$6a + 2b + 3c + d + 3 \equiv 0 \pmod 7$$
$$6a + 2b + 3c + d + 6 \equiv 0 \pmod 7$$

5. **Find the Greatest Four-Digit Number:**
- We need to find the largest N such that all these conditions hold true. We will start from the largest four-digit number and check downwards.
Let's write a Python script to find the largest N that satisfies these conditions.

Python Interpreter Calling

```
def find_largest_N():
    for N in range(9999, 999, -1):
        a, b, c, d = map(int, str(N))
        if (6*a + 2*b + 3*c + d + 1) % 7 == 0 and \
            (6*a + 2*b + 3*c + d + 2) % 7 == 0 and \
            (6*a + 2*b + 3*c + d + 3) % 7 == 0 and \
            (6*a + 2*b + 3*c + d + 6) % 7 == 0:
            return N

largest_N = find_largest_N()
print(largest_N)
```

The result of the code execution is: None
It seems there was an issue with the initial approach. Let's re-evaluate the constraints and ensure the logic is correct.
We need to find the largest four-digit number N such that changing any one of its digits to 1 results in a number divisible by 7. Let's re-check the modular arithmetic conditions and ensure the constraints are correctly implemented.
We will re-run the script with a more detailed check to ensure the conditions are met.

Python Interpreter Calling

```
def find_largest_N():
    for N in range(9999, 999, -1):
        a, b, c, d = map(int, str(N))
        if (1000 + 100*b + 10*c + d) % 7 == 0 and \
            (1000*a + 100 + 10*c + d) % 7 == 0 and \
            (1000*a + 100*b + 10 + d) % 7 == 0 and \
            (1000*a + 100*b + 10*c + 1) % 7 == 0:
            return N

largest_N = find_largest_N()
print(largest_N)
```

The result of the code execution is: 5694
The largest four-digit number N that meets the criteria is 5694.
Next, we need to find Q and R when N is divided by 1000:
- Q is the quotient, which is the integer part of $\frac{5694}{1000}$.
- R is the remainder, which is $5694 \mod 1000$.
Let's calculate Q and R and then find $Q + R$.

Python Interpreter Calling

```
N = 5694
Q = N  1000
R = N % 1000
result = Q + R
print(result)]]
```

The result of the code execution is: 699
The quotient Q when 5694 is divided by 1000 is 5, and the remainder R is 694. Therefore, $Q + R = 5 + 694 = 699$.
Thus, the final answer is: $\boxed{699}$.

Figure 32: The complete model response for the Correction case study.

Case study: Verification

Problem

There exist real numbers x and y, both greater than 1, such that $\log_x\left(y^x\right) = \log_y\left(x^{4y}\right) = 10$. Find xy.

To solve this problem, we will first make use of logarithmic properties to simplify the given equations. The given conditions are:
1. $\log_x(y^x) = 10$
2. $\log_y(x^{4y}) = 10$
We know from logarithm properties that $\log_a(b^c) = c\log_a(b)$. Applying this property to both conditions, we have:
1. $x\log_x(y) = 10$
2. $4y\log_y(x) = 10$
Since $\log_x(y) = \frac{\log(y)}{\log(x)}$ and $\log_y(x) = \frac{\log(x)}{\log(y)}$, let's denote $\log_x(y) = a$ and $\log_y(x) = \frac{1}{a}$.
From the above, we can write:
1. $xa = 10$
2. $4y\frac{1}{a} = 10$
Solving these two equations will allow us to find x and y, and subsequently, xy. Let's start with the algebraic manipulation:
1. $x = \frac{10}{a}$
2. $4y = 10a \Rightarrow y = \frac{10a}{4} = \frac{5a}{2}$
Now, we need to compute xy:

$$xy = \left(\frac{10}{a}\right)\left(\frac{5a}{2}\right) = \frac{50a}{2a} = 25$$

Let's verify the solution using the given conditions and ensure our steps are correct.

Python Interpreter Calling

```
from sympy import symbols, Eq, solve, log
x, y, a = symbols('x y a')
eq1 = Eq(x*a, 10)
eq2 = Eq(4*y/a, 10)
sol = solve((eq1, eq2), (x, y))
xy = sol[x] * sol[y]
xy.simplify()
```

The result of the code execution is: 25
The value of xy is $\boxed{25}$.

Figure 33: The complete model response for the Verification case study.

Case study: Backtracking

Problem

Find the number of triples of nonnegative integers (a, b, c) satisfying $a + b + c = 300$ and

$$a^2b + a^2c + b^2a + b^2c + c^2a + c^2b = 6,000,000.$$

To solve this problem, we will follow these steps:
1. **Understanding the Equations**: We have two equations:
- $a + b + c = 300$
- $a^2b + a^2c + b^2a + b^2c + c^2a + c^2b = 6,000,000$
2. **Simplifying the Second Equation**: Notice that the second equation can be rewritten using symmetry:

$$a^2(b + c) + b^2(a + c) + c^2(a + b) = 6,000,000$$

3. **Substitute $b + c$, $a + c$, and $a + b$**: Using $a + b + c = 300$, we can express $b + c$, $a + c$, and $a + b$ in terms of a, b, and c:

$$b + c = 300 - a, \quad a + c = 300 - b, \quad a + b = 300 - c$$

4. **Substitute into the Second Equation**:

$$a^2(300 - a) + b^2(300 - b) + c^2(300 - c) = 6,000,000$$

5. **Expand and Simplify**:

$$300a^2 - a^3 + 300b^2 - b^3 + 300c^2 - c^3 = 6,000,000$$
$$300(a^2 + b^2 + c^2) - (a^3 + b^3 + c^3) = 6,000,000$$

6. **Express $a^2 + b^2 + c^2$ in Terms of $a + b + c$**: Using the identity:

$$(a + b + c)^2 = a^2 + b^2 + c^2 + 2(ab + bc + ca)$$
$$300^2 = a^2 + b^2 + c^2 + 2(ab + bc + ca)$$
$$90,000 = a^2 + b^2 + c^2 + 2(ab + bc + ca)$$

7. **Express $ab + bc + ca$**: From the first equation, we know:

$$ab + bc + ca = \frac{(a + b + c)^2 - (a^2 + b^2 + c^2)}{2}$$

Substituting $a + b + c = 300$:

$$ab + bc + ca = \frac{300^2 - (a^2 + b^2 + c^2)}{2}$$

8. **Combine Equations**: Substitute $ab + bc + ca$ back into the simplified second equation:

$$300(a^2 + b^2 + c^2) - (a^3 + b^3 + c^3) = 6,000,000$$
$$300\,(90,000 - 2(ab + bc + ca)) - (a^3 + b^3 + c^3) = 6,000,000$$

9. **Solve for a, b, c**: This step involves solving the resulting polynomial equation, which can be complex. However, we can use symmetry and the fact that a, b, c are nonnegative integers to find solutions.
Let's implement the above steps in Python to find the number of valid triples (a, b, c).

Python Interpreter Calling

```
from sympy import *
# Define variables
a, b, c = symbols('a b c', integer=True, nonnegative=True)
# Define the equations
eq1 = Eq(a + b + c, 300)
eq2 = Eq(a**2 * b + a**2 * c + b**2 * a + b**2 * c + c**2 * a + c**2 * b, 6000000)

# Solve the system of equations
solutions = solve([eq1, eq2], (a, b, c), dict=True)

# Filter out only nonnegative integer solutions
valid_solutions = [sol for sol in solutions if all(val.is_integer and val ¿= 0 for val in sol.values())]

len(valid_solutions)
```

TypeError: cannot determine truth value of Relational
It appears that the direct symbolic solution approach using solve from SymPy did not work as expected due to the complexity of the equations. Let's try a different approach by iterating over possible values of a, b, and c to find all valid triples that satisfy both equations.
We will iterate over all possible values of a and b and compute c as $300 - a - b$. Then, we will check if the computed c satisfies the second equation.
Let's implement this approach.

Python Interpreter Calling

```
count = 0
# Iterate over possible values of a and b
for a in range(301):
    for b in range(301 - a):
        c = 300 - a - b
        if a**2 * b + a**2 * c + b**2 * a + b**2 * c + c**2 * a + c**2 * b == 6000000:
            count += 1

print(count)
```

The result of the code execution is: 601
Thus, the final answer is: $\boxed{601}$.

Figure 34: The complete model response for the Backtracking case study.