AlphaApollo: Orchestrating Foundation Models and Professional Tools into a Self-Evolving 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], Linrui Xu[1], Tian Cheng[1], Guanyu Jiang[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, a self-evolving agentic reasoning system that aims to address two bottlenecks in foundation model (FM) reasoning—limited model-intrinsic capacity and unreliable test-time iteration. AlphaApollo orchestrates multiple models with professional tools to enable deliberate, verifiable reasoning. It couples (i) a computation tool (Python with numerical and symbolic libraries) and (ii) a retrieval tool (task-relevant external information) to execute exact calculations and ground decisions. The system further supports multi-round, multi-model solution evolution via a shared state map that records candidates, executable checks, and feedback for iterative refinement. In evaluations on AIME 2024/2025 across multiple models, AlphaApollo delivers consistent gains: +5.15% Average@32 and +23.34% Pass@32 for Qwen2.5-14B-Instruct, and +8.91% Average@32 with +26.67% Pass@32 for Llama-3.3-70B-Instruct. Tool-use analysis shows that more than 80% of tool calls are successfully executed, with consistent outperformance of non-tool baselines, thereby lifting the capability ceiling of FMs. More empirical results and implementation details will be updated at `https://github.com/tmlr-group/AlphaApollo`. [1]

1 Introduction

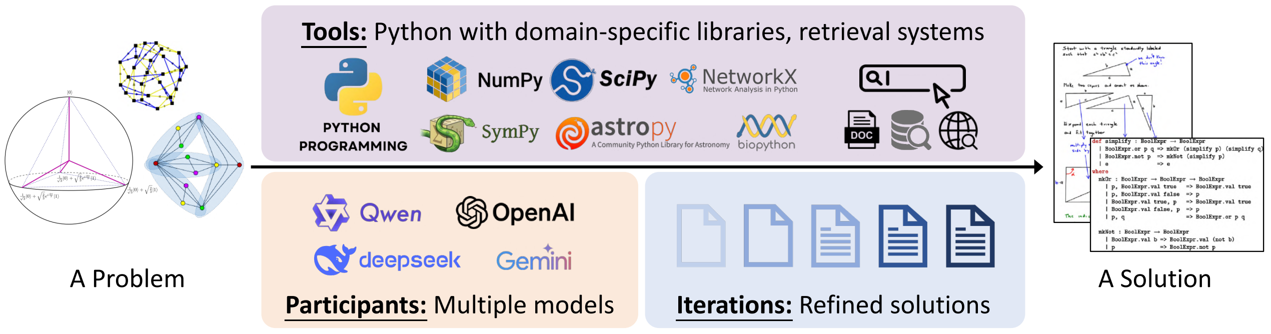
Foundation models (FMs) increasingly boost diverse applications through explicit *reasoning*, decomposing complex tasks into steps for more reliable decisions (Li et al., 2025). Yet even state-of-the-art models struggle on challenging benchmarks: as of October 2025, GPT-5 and Gemini 2.5 Pro reach only 25.3% and 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), with open-source models lagging further. Beyond math and code, limited computational power and insufficient domain knowledge constrain their impact in domains such as biology, chemistry, and healthcare, undermining reliability for real-world scientific use.

The two key bottlenecks in FM reasoning are *(i) model-intrinsic capacity* and *(ii) test-time iteration*. First, prompting and post-training methods ultimately depend on the base model's capacity, raising the question of whether observed gains reflect genuinely *emergent* reasoning or improved elicitation. Behaviors such as self-reflection ("aha" moments) largely originate in pre-training and can be further elicited, whereas abilities requiring exact calculus or symbolic manipulation remain constrained by next-token prediction (Yang et al., 2024c; Wang et al., 2025). Second, effective

[1]This project is ongoing. We welcome feedback from the community and will frequently update our technical report.

Tools: Space suit, rocket, spacecraft, etc.
The Earth
Participants: 400k+ people
Iterations: Apollo 1 to Apollo 17
The Moon

(a) The Apollo Program (in 1960s) for moon landing with humans


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

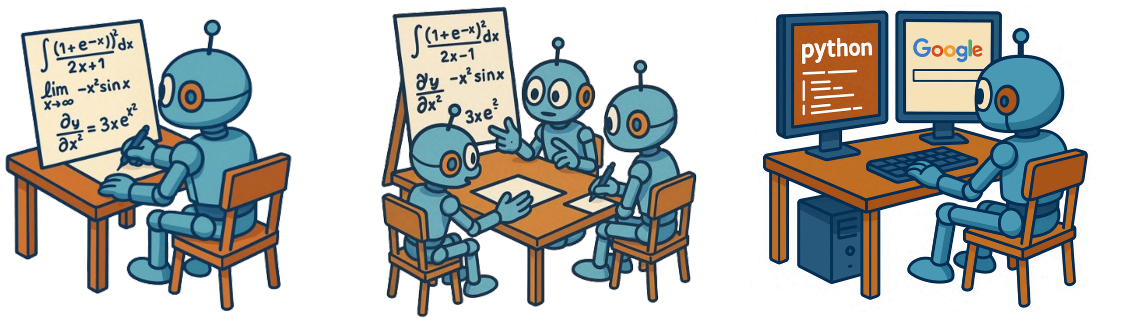
(b) The AlphaApollo System (ours) for problem solving with foundation models

Figure 1: The *Apollo Program* mobilized more than 400,000 people over the span of a decade, from Apollo 1 to Apollo 17, to develop the mission systems that enabled humanity's first moon landings. As a tribute to this historic achievement, we name our project *AlphaApollo*. The two projects share common principles despite their different domains: (1) reliance on advanced **tools**—the Apollo mission system and, in our case, Python code with retrieval systems; (2) collaboration among many **participants**—people or models; and (3) **iterations** across a series of missions or solutions.

test-time reasoning often traverses a large solution space with trustworthy feedback. Yet, current scaling strategies—parallel, sequential, or hybrid—heavily rely on the model's own signals, which are subjective and unreliable (Gao et al., 2025). In the absence of ground-truth verification, reliably assessing solution quality at inference time remains a significant challenge. Moreover, extending iteration to multiple interacting models incurs additional compute and coordination overhead while often yielding little or no benefit over non-interactive single-model scaling.

This work introduces *AlphaApollo*, an agentic reasoning *system* designed to overcome these bottlenecks. Its design principle is to *orchestrate* diverse models and professional tools into a *self-evolving system*, enabling parallel, deep, agentic reasoning. Inspired by the Apollo program of the 1960s (Figure 1(a)), AlphaApollo adopts a strategy of setting clear goals, concentrating expertise and resources, and coordinating systematic collaboration under shared trust and organizational support, making it possible to tackle complex problems once deemed impossible. To transcend the capacity limits of a single model, AlphaApollo centers on two core features, detailed as follows.

- First, AlphaApollo couples two classes of professional tools—*(i) computation* and *(ii) retrieval*—to enable deliberate, tool-augmented reasoning. The computation tool is a Python interpreter with domain libraries (e.g., SciPy (Virtanen et al., 2020) and SymPy (Meurer et al., 2017)) for numerical and symbolic calculations. The retrieval tool surfaces task-relevant information from external sources (e.g., library documentation or search engines), such as the usage of a specific SciPy function. This agentic paradigm departs from single-model and multi-model paradigms by integrating exact computation with targeted retrieval to strengthen FM reasoning (Figure 2).



(a) single-model reasoning (b) multi-model reasoning (c) agentic reasoning (AlphaApollo)

Figure 2: A comparison of three reasoning paradigms.

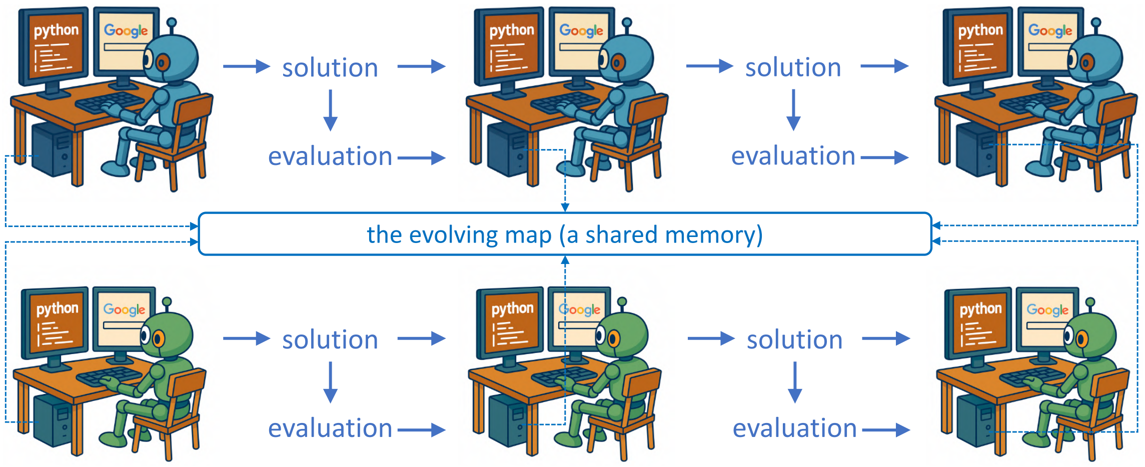


Figure 3: Test-time evolving with multiple models.

- Second, AlphaApollo enables multi-round, multi-model solution evolution (Figure 3). Each model has full access to the toolset to propose candidate solutions and perform follow-up evaluation. For example, solving a math problem may produce a textual derivation with Python code; the code is then executed and tested to yield verifiable, fine-grained feedback that guides further refinement. All candidates and evaluations are recorded as states in an evolving map (as a shared memory), over which all models operate in parallel, referencing prior states to generate refined solutions.

Empirically, we evaluate AlphaApollo on mathematics reasoning benchmarks, AIME 2024 and 2025, across multiple model families, including Qwen2.5 (7B/14B/32B/72B), Qwen3-235B-A22B, and Llama-3.3-70B-Instruct. Across all models, AlphaApollo yields consistent gains—up to +9.16% Average@32 and +23.34% Pass@32 on medium Qwen2.5 models; substantial improvements on Llama3.3-70B-Instruct for AIME 2025 (+16.67% Average@32; Pass@32 increasing from 23.33% to 46.67%); and stable gains even on the strong Qwen3-235B-A22B—demonstrating effective scalability. Tool-use analysis further shows that tool-call correctness is near or above 80% for most settings and that tool-augmented responses consistently outperform non-tool ones, indicating that AlphaApollo both improves average performance and raises the capability ceiling for complex reasoning.

AlphaApollo is under active development. This technical report will be updated as new features arrive. The current release introduces our first feature: tool-augmented reasoning. Next, we will add the second feature of test-time scaling—evolving solutions with single or multiple models—followed by broader integration of frontier models, professional tools, and advanced algorithms. We will open-source the full code and experimental results to enable reproducibility and extension.

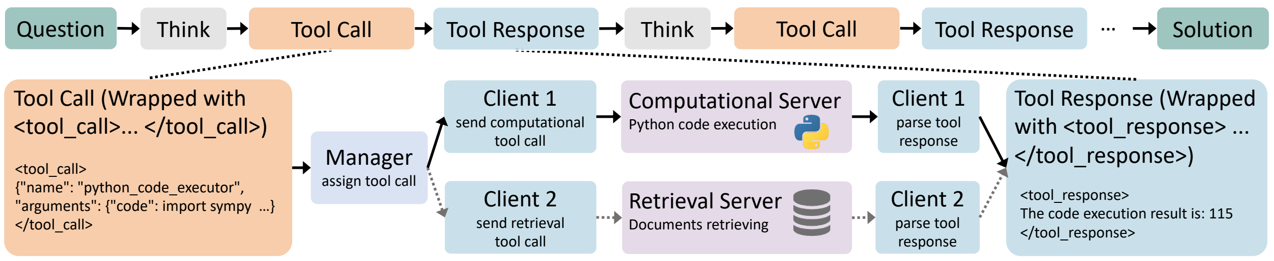


Figure 4: Schematic illustrating the tool-calling process of the rollout framework. The example shown demonstrates computational tool calls, while the dashed path indicates retrieval tool calls.

2 AlphaApollo

This section presents the technical design of AlphaApollo. Section 2.1 introduces the rollout framework, followed by the computation and retrieval modules in Sections 2.2 and 2.3, respectively.

Tool-augmented reasoning. As illustrated in Figure 4, AlphaApollo structures the reasoning pipeline (of a single model) into three types of tokens: *think*, *tool call*, and *tool response*. Specifically, the *think* tokens represent the model's internal reasoning process. Whenever the model requires external support—computational or informational—it issues a *tool call*, which is intercepted and executed by the AlphaApollo system. The resulting *tool response* is then inserted back into the model's context, allowing the model's reasoning to resume. Through iterative cycles of thinking, tool calls, and tool responses, the reasoning proceeds until a final solution (with an answer) is generated.

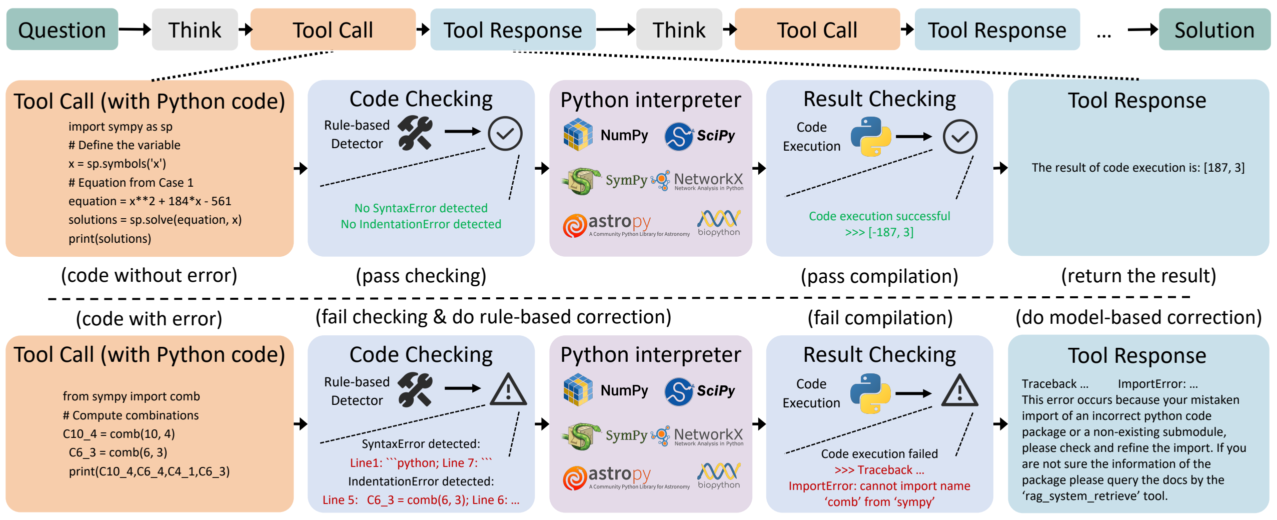
2.1 The Rollout Framework

In AlphaApollo, the reasoning trajectory generated by a particular model encompasses both **model inference** (within the *think* and *tool call* tokens) and **tool execution** (within the *tool response* tokens). This tool-augmented reasoning trajectory is generated by the rollout framework, which bridges the gap between a foundation model and multiple functional tools. Specifically:

- **For model inference,** AlphaApollo can generate the next token locally or remotely, with a particular model. The supported *inference backends* include SGLang (Zheng et al., 2024), vLLM (Kwon et al., 2023), HuggingFace Transformers (Huggingface, 2025), and external APIs (OpenAI, 2025).
- **For tool execution,** AlphaApollo adopts the *Model Context Protocol (MCP)* (Anthropic, 2024), which standardizes how tools provide context to foundation models. Through MCP, AlphaApollo seamlessly integrates diverse tools and foundation models under a unified protocol.

Following the MCP's architecture, AlphaApollo's rollout framework consists of one *manager*, two *clients*, and two *servers*, as illustrated in Figure 4. Here, each computational or retrieval tool is paired with a *server* and a *client*. Specifically:

- **Functions.** The *manager* coordinates the overall rollout, assigning a tool call to a *client*. The *client* monitors tool-call status and results, communicating directly with the *server*. The *server* executes the tool call and delivers the result to the *client*, which returns the tool response to the model.
- **Running-time inference.** When the model generates a tool call enclosed within `<tool_call>` and `</tool_call>` tags, the *manager* captures it. The *manager* then assigns the call to the appropriate *client*, which forwards it to the designated *server*. The *server* executes the requested task and



```
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)
```

```
No SyntaxError detected
No IndentationError detected
```

```
Code execution successful
>>> [-187, 3]
```

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

```
from sympy import comb
# Compute combinations
C10_4 = comb(10, 4)
 C6_3 = comb(6, 3)
 print(C10_4,C6_4,C4_1,C6_3)
```

```
SyntaxError detected:
Line1: ```python; Line 7: ```
IndentationError detected:
Line 5:   C6_3 = comb(6, 3); Line 6: ...
```

```
Code execution failed
>>> Traceback ...
ImportError: cannot import name
'comb' from 'sympy'
```

Traceback …   ImportError: …
This error occurs because your mistaken import of an incorrect python code package or a non-existing submodule, please check and refine the import. If you are not sure the information of the package please query the docs by the 'rag_system_retrieve' tool.

Figure 5: The pipeline of the computational module in processing code with or without errors. The light-blue components (Code checking, Result checking) correspond to Client 1, while the purple components correspond to the Computational Server.

returns the result to the *client*. The *client* extracts execution results, parses potential errors, and delivers them back to the model, wrapped within `<tool_response>` and `</tool_response>` tags.

- **Initialization.** The *manager* loads two configuration files: (1) a server configuration file specifying connection details such as URLs and authentication, and (2) a tool configuration file defining tool-level parameters such as rate limits, timeouts, and tool names. The *manager* uses the server configuration to launch *servers*, and the tool configuration to initialize *clients*. In addition, the model should know all the available tools before reasoning. This is achieved by the tool schemas that define the accessible tools and their usage (details in Appendix A.2), including the tool names, descriptions, and input parameters, together with the system prompt through the chat template.

AlphaApollo guides the model to decide when to invoke a computational or retrieval tool through prompting: the system provides descriptions of available tools and requests the model to determine when and how to call them. Future versions will optimize it via post-training for adaptive tool use.

2.2 The Computational Module

This module integrates a Python interpreter with several external libraries of Python, as illustrated in Figure 5. 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.

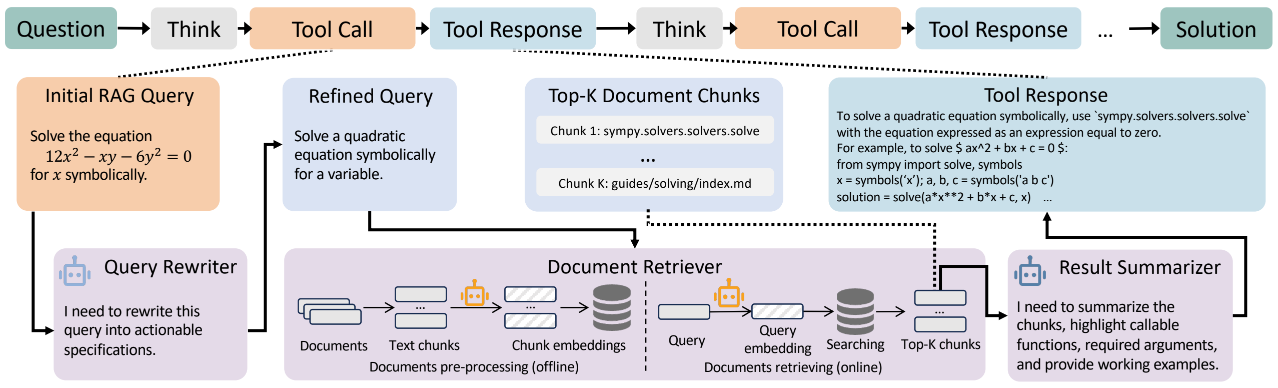
Error correction. The model-generated code can contain errors. To improve the robustness against these errors, AlphaApollo's computational module incorporates a hybrid error-correction mechanism that combines rule-based and model-based approaches. When the model generates Python code, the rule-based approach detects and automatically corrects errors identifiable through predefined rules, and then passes the corrected code to the Python interpreter for execution. Then, if a run-time error arises, the model-based approach analyzes the execution results to identify potential causes and propose targeted correction strategies, thereby guiding the model toward improved subsequent code generation. We outline the specific error types addressed by each approach (more in Appendix B.1):

- **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.
- **Model-based error correction.** When run-time errors such as `NameError` or `IndexError` occur, this approach provides detailed feedback derived from the execution results, including likely causes and suggested fixes, to guide the model in self-correction. For instance, in the case of a `NameError`, the feedback may include guidance: "This error is caused by using a module or a variable that is not defined, please check if you have imported all using Python code packages in your code, and you write complete code in your current tool calling."

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 (Section 2.3) 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.

2.3 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



Solve the equation $12x^2 - xy - 6y^2 = 0$ for x symbolically.

Solve a quadratic equation symbolically for a variable.

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)    ...
```

I need to rewrite this query into actionable specifications.

I need to summarize the chunks, highlight callable functions, required arguments, and provide working examples.

Figure 6: 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.

Figure 6, 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. Details are provided in Appendix A.3.

- **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 sentence-level 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 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)`. The summarizer is implemented using an instruction-following model with a specialized prompt template, which is provided in Appendix A.3.

Table 1: The detailed settings of models in our experiments. Here, output length refers to the maximum response length specified during evaluation; model type characterizes the reasoning paradigm—*instruction models* directly produce answers that may include explicit multi-step reasoning, while *reasoning models* first generate extended internal reasoning traces before presenting a structured multi-step solution; and architecture indicates whether the model adopts a Mixture-of-Experts (MoE) or a dense structure.

Model Family	**Model Name**	Maximum context length	Output length	Model type	Architecture
Qwen2.5 (Yang et al., 2024a)	Qwen2.5-7B-Instruct	128K	8k	Instruction	Dense
	Qwen2.5-14B-Instruct	128K	8k	Instruction	Dense
	Qwen2.5-32B-Instruct	128K	8k	Instruction	Dense
	Qwen2.5-72B-Instruct	128K	8k	Instruction	Dense
Qwen3 (Yang et al., 2025)	Qwen3-235B-A22B	80k	32k	Reasoning	MoE
Llama (Grattafiori et al., 2024)	Llama3.3-70B-Instruct	128k	8k	Instruction	Dense

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.

3 Experiments

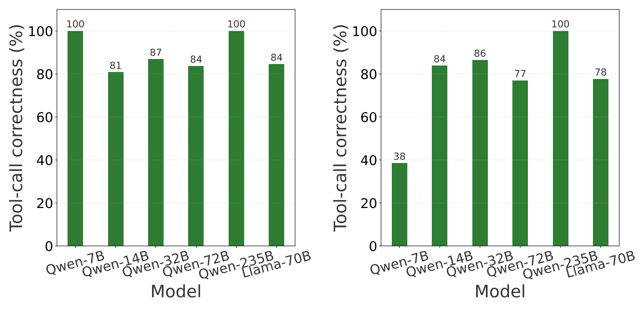
In this section, we evaluate the AlphaApollo on complex reasoning tasks. We first describe the experimental setup, followed by the main experimental results and detailed analysis.

Experiment settings. We evaluate AlphaApollo across different model families, including the Qwen2.5 series, Qwen3 series, and Llama series. The model configurations are shown in Table 1. For benchmarks, we use AIME 2024 and AIME 2025, both derived from the American Invitational Mathematics Examination, a competition widely considered one of the most challenging mathematics competitions at the high-school level. These benchmarks evaluate sophisticated problem-solving and reasoning abilities that exceed conventional competition datasets. The two iterations, AIME 2024 and 2025, have distinct problem sets, facilitating annual assessment of robustness and generalization. For the experimental setup, we evaluate all models with sampling parameters of temperature = 0.6, top-k = 20, and top-p = 0.95 for evaluation. Each question is sampled 32 times to mitigate the impact of randomness. We report both Average@32 and Pass@32 results: the former reflects the model's average performance, while the latter indicates the upper bound of the model's capability.

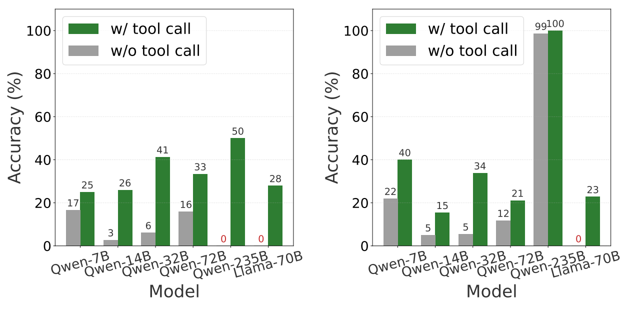
Main results. We summarize the following observations from the experimental results in Table 2. The results demonstrate that AlphaApollo consistently improves performance across different model families and scales. For medium-sized models such as Qwen2.5-14B/32B/72B-Instruct, AlphaApollo achieves substantial gains in both Average@32 and Pass@32, with improvements of up to 9.16 % and 23.34 %, respectively. Notably, for the large model Qwen3-235B-A22B, although the base model already performs strongly, AlphaApollo still provides consistent improvements, showing that our method scales effectively. Moreover, on Llama3.3-70B-Instruct, AlphaApollo delivers remarkable boosts—especially on AIME 2025, where Average@32 increases by 16.67% and Pass@32 doubles from 23.33% to 46.67%. Overall, these results indicate that AlphaApollo can significantly strengthen both the average performance and the capability ceiling of state-of-the-art FMs.

Table 2: Main experiment results (in %). The **boldface** numbers denote the best results. We also report the absolute improvement of AlphaApollo over the base model for each dataset.

Base Model	Setting	Datasets				Average	
		AIME 2024		AIME 2025			
		Avg@32	Pass@32	Avg@32	Pass@32	Avg@32	Pass@32
Qwen2.5-7B-Instruct	Base	**11.87**	40.00	7.08	33.33	9.48	36.67
	AlphaApollo (ours)	11.15	**46.67**	**9.06**	**46.67**	**10.11**	**46.67**
		(0.72↓)	(6.67↑)	(1.98↑)	(13.34↑)	(0.63↑)	(10.00↑)
Qwen2.5-14B-Instruct	Base	14.17	43.33	13.23	36.67	13.70	40.00
	AlphaApollo (ours)	**23.33**	**66.67**	**14.37**	**60.00**	**18.85**	**63.34**
		(9.16↑)	(23.34↑)	(1.14↑)	(23.33↑)	(5.15↑)	(23.34↑)
Qwen2.5-32B-Instruct	Base	18.33	40.00	12.60	43.33	15.47	41.67
	AlphaApollo (ours)	**23.75**	**63.33**	**19.27**	**53.33**	**21.51**	**58.33**
		(5.42↑)	(23.33↑)	(6.67↑)	(10.00↑)	(6.04↑)	(16.66↑)
Qwen2.5-72B-Instruct	Base	18.23	50.00	13.75	30.00	15.99	40.00
	AlphaApollo (ours)	**22.81**	**56.67**	**14.37**	**53.33**	**18.59**	**55.00**
		(4.58↑)	(6.67↑)	(0.62↑)	(23.33↑)	(2.60↑)	(15.00↑)
Qwen3-235B-A22B	Base	83.44	90.00	79.90	96.67	81.67	93.34
	AlphaApollo (ours)	**86.67**	**96.67**	**84.58**	**96.67**	**85.63**	**96.67**
		(3.23↑)	(6.67↑)	(4.68↑)	(0.00↑)	(3.96↑)	(3.33↑)
Llama3.3-70B-Instruct	Base	26.56	40.00	5.10	23.33	15.83	31.67
	AlphaApollo (ours)	**27.71**	**70.00**	**21.77**	**46.67**	**24.74**	**58.34**
		(1.15↑)	(30.00↑)	(16.67↑)	(23.34↑)	(8.91↑)	(26.67↑)



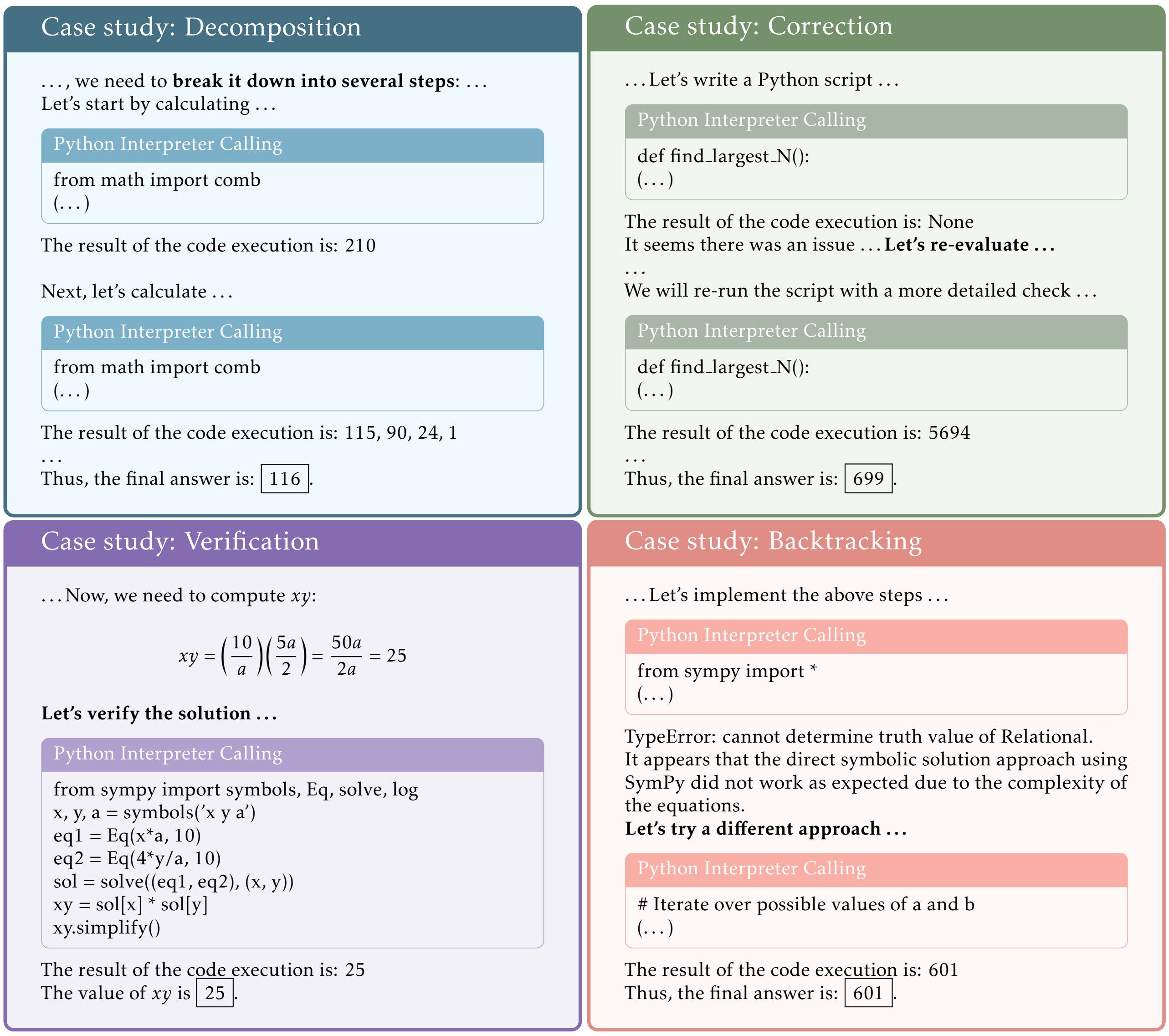
(a) Tool-call correctness (left: AIME24, right: AIME25)



(b) Accuracy (left: AIME24, right: AIME25)

Figure 7: (a) **Tool-call correctness.** Most models use tools correctly in nearly or more than 80% of cases. (b) **Comparison between responses with or without tool calls.** Note that to eliminate the differences between questions, we only collect the questions where at least one response includes a tool call and at least one response does not.

Analysis. We analyze our system from two perspectives: the quality of tool calls and the impact of tools on final performance. For **the quality of tool calls**, we extract tool responses from model outputs and check for tool-call errors, such as execution errors in the computational module, querying the retrieval module for nonexistent packages, or attempts to call nonexistent tools. As shown in Figure 7a, most models use tools correctly in more than 80% of cases, indicating our framework effectively guides models to generate correct tool calls. For **the impact of tools on performance**, we divide the model outputs into two types: those with tool calls and those without. For questions with both types of responses, we compute accuracies separately. As shown in Figure 7b, responses with tool calls consistently outperform those without. Notably, with the assistance of tools, Qwen3-235B-A22B and Llama3.3-70B-Instruct are able to solve problems that they completely fail on without tool support, indicating that tools expand the models' capability frontier.



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 8: Case studies illustrating models' diverse cognitive behaviors exhibited by AlphaApollo. We present the complete model responses in Appendix B.2.

Case studies. With the support of AlphaApollo's professional tools, models exhibit diverse cognitive behaviors in their reasoning processes. These behaviors originate from the pre-training data of the base model, which equips the model with the ability to conduct deliberate reasoning (Gandhi et al., 2025). We summarize these cognitive behaviors in tool-augmented reasoning as follows:

- **Decomposition**. The model demonstrates the ability to break 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 Related Work

In this section, we systematically review prior work related to our three key features: tool-integrated reasoning in Section 4.1, multi-model reasoning in Section 4.2, and test-time iteration in Section 4.3. 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.

4.1 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 pre-defined calling stages. For instance, SciMaster (Chai et al., 2025a) 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., 2025b), 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.

4.2 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.

4.3 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.

5 Conclusion

We introduce AlphaApollo, a self-evolving agentic reasoning system that aims to tackle two core bottlenecks in foundation model reasoning—limited model-intrinsic capacity and unreliable, compute-intensive test-time iteration—by orchestrating multiple models with professional tools. By pairing exact computation (Python with scientific and symbolic libraries) with targeted retrieval, AlphaApollo produces verifiable, tool-augmented solutions, while a shared, evolving state enables parallel proposal, execution, testing, and refinement. Empirically, it delivers consistent improvements across AIME 2024/2025 and diverse model families. This release focuses on tool-augmented reasoning; upcoming versions will add multi-round, multi-model test-time scaling, and extensions to broader domains. We will open-source the full system and results to support reproducibility and accelerate community-driven progress toward trustworthy, real-world reasoning.

References

Agashe, S., Fan, Y., Reyna, A., and Wang, X. E. Llm-coordination: evaluating and analyzing multi-agent coordination abilities in large language models. *arXiv preprint arXiv:2310.03903*, 2023.

Anthropic. Introducing the model context protocol, 2024. URL `https://www.anthropic.com/news/model-context-protocol`.

Asai, A., Wu, Z., Wang, Y., Sil, A., and Hajishirzi, H. Self-RAG: Learning to retrieve, generate, and critique through self-reflection. In *ICLR*, 2024.

Chai, J., Tang, S., Ye, R., Du, Y., Zhu, X., Zhou, M., Wang, Y., E, W., Zhang, Y., Zhang, L., and Chen, S. 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*, 2025a.

Chai, J., Yin, G., Xu, Z., Yue, C., Jia, Y., Xia, S., Wang, X., Jiang, J., Li, X., Dong, C., et al. Rlfactory: A plug-and-play reinforcement learning post-training framework for llm multi-turn tool-use. *arXiv preprint arXiv:2509.06980*, 2025b.

Chang, K., Shi, Y., Wang, C., Zhou, H., Hu, C., Liu, X., Luo, Y., Ge, Y., Xiao, T., and Zhu, J. Step-level verifier-guided hybrid test-time scaling for large language models. *arXiv preprint arXiv:2507.15512*, 2025.

Chen, B., Shu, C., Shareghi, E., Collier, N., Narasimhan, K., and Yao, S. Fireact: Toward language agent fine-tuning. *arXiv preprint arXiv:2310.05915*, 2023a.

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

Chen, X., Lin, M., Schärli, N., and Zhou, D. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023c.

Chollet, F., Knoop, M., Kamradt, G., Landers, B., and Pinkard, H. Arc-agi-2: A new challenge for frontier ai reasoning systems. *arXiv preprint arXiv:2505.11831*, 2025.

Du, Y., Li, S., Torralba, A., Tenenbaum, J. B., and Mordatch, I. Improving factuality and reasoning in language models through multiagent debate. In *ICML*, 2024.

Edge, D., Trinh, H., Cheng, N., Bradley, J., Chao, A., Mody, A., Truitt, S., Metropolitansky, D., Ness, R. O., and Larson, J. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024.

Feng, L., Xue, Z., Liu, T., and An, B. Group-in-group policy optimization for llm agent training. In *NeurIPS*, 2025.

Gandhi, K., Chakravarthy, A., Singh, A., Lile, N., and Goodman, N. D. Cognitive behaviors that enable self-improving reasoners, or, four habits of highly effective stars. In *COLM*, 2025.

Gao, H.-a., Geng, J., Hua, W., Hu, M., Juan, X., Liu, H., Liu, S., Qiu, J., Qi, X., Wu, Y., et al. A survey of self-evolving agents: On path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*, 2025.

Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. Pal: Program-aided language models. In *ICML*, 2023a.

Gao, S., Shi, Z., Zhu, M., Fang, B., Xin, X., Ren, P., Chen, Z., Ma, J., and Ren, Z. Confucius: Iterative tool learning from introspection feedback by easy-to-difficult curriculum. In *AAAI*, 2024.

Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Wang, H., and Wang, H. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2023b.

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. Array programming with NumPy. *Nature*, 2020.

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., Wang, J., Wang, Z., Yau, S. K. S., Lin, Z., et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *ICLR*, 2024.

Huang, J., Chen, X., Mishra, S., Zheng, H. S., Yu, A. W., Song, X., and Zhou, D. Large language models cannot self-correct reasoning yet. In *ICLR*, 2024.

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

Jiang, D., Lu, Y., Li, Z., Lyu, Z., Nie, P., Wang, H., Su, A., Chen, H., Zou, K., Du, C., et al. Verltool: Towards holistic agentic reinforcement learning with tool use. *arXiv preprint arXiv:2509.01055*, 2025.

Kim, G., Baldi, P., and McAleer, S. Language models can solve computer tasks. In *NeurIPS*, 2023.

Knappe, T., Li, R., Chauhan, A., Chhua, K., Zhu, K., and O'Brien, S. Semantic self-consistency: Enhancing language model reasoning via semantic weighting. *arXiv preprint arXiv:2410.07839*, 2024.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *SOSP*, 2023.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *NeurIPS*, 2020.

Li, Y., Du, Y., Zhang, J., Hou, L., Grabowski, P., Li, Y., and Ie, E. Improving multi-agent debate with sparse communication topology. *arXiv preprint arXiv:2406.11776*, 2024.

Li, Z.-Z., Zhang, D., Zhang, M.-L., Zhang, J., Liu, Z., Yao, Y., Xu, H., Zheng, J., Wang, P.-J., Chen, X., et al. From system 1 to system 2: A survey of reasoning large language models. *arXiv preprint arXiv:2502.17419*, 2025.

Liu, Z., Hoang, T., Zhang, J., Zhu, M., Lan, T., Kokane, S., Tan, J., Yao, W., Liu, Z., Feng, Y., Murthy, R., Yang, L., Savarese, S., Niebles, J. C., Wang, H., Heinecke, S., and Xiong, C. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. *arXiv preprint arXiv:2406.18518*, 2024.

Lu, P., Chen, B., Liu, S., Thapa, R., Boen, J., and Zou, J. Octotools: An agentic framework with extensible tools for complex reasoning. *arXiv preprint arXiv:2502.11271*, 2025.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al. Self-refine: Iterative refinement with self-feedback. In *NeurIPS*, 2023.

Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., et al. Sympy: symbolic computing in python. *PeerJ Computer Science*, 2017.

Muennighoff, N., Yang, Z., Shi, W., Li, X. L., Fei-Fei, L., Hajishirzi, H., Zettlemoyer, L., Liang, P., Candès, E., and Hashimoto, T. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*, 2025.

Naik, R., Chandrasekaran, V., Yuksekgonul, M., Palangi, H., and Nushi, B. Diversity of thought improves reasoning abilities of llms. *arXiv preprint arXiv:2310.07088*, 2023.

Nakano, R., Hilton, J., Balaji, S., Wu, J., Ouyang, L., Kim, C., Hesse, C., Jain, S., Kosaraju, V., Saunders, W., et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.

Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J., Mehrabian, A., 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.

Paul, D., Ismayilzada, M., Peyrard, M., Borges, B., Bosselut, A., West, R., and Faltings, B. Refiner: Reasoning feedback on intermediate representations. *arXiv preprint arXiv:2304.01904*, 2023.

Phan, L., Gatti, A., Han, Z., Li, N., Hu, J., Zhang, H., Zhang, C. B. C., Shaaban, M., Ling, J., Shi, S., et al. Humanity's last exam. *arXiv preprint arXiv:2501.14249*, 2025.

Press, O., Zhang, M., Min, S., Schmidt, L., Smith, N. A., and Lewis, M. Measuring and narrowing the compositionality gap in language models. *arXiv preprint arXiv:2210.03350*, 2022.

Qi, J., Ye, X., Tang, H., Zhu, Z., and Choi, E. Learning to reason across parallel samples for llm reasoning. *arXiv preprint arXiv:2506.09014*, 2025.

Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., Lin, Y., Cong, X., Tang, X., Qian, B., Zhao, S., Hong, L., Tian, R., Xie, R., Zhou, J., Gerstein, M., Li, D., Liu, Z., and Sun, M. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *ICLR*, 2024.

Qiu, J., Qi, X., Zhang, T., Juan, X., Guo, J., Lu, Y., Wang, Y., Yao, Z., Ren, Q., Jiang, X., et al. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *arXiv preprint arXiv:2505.20286*, 2025.

Qu, Y., Zhang, T., Garg, N., and Kumar, A. Recursive introspection: Teaching language model agents how to self-improve. In *NeurIPS*, 2024.

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

Shang, N., Liu, Y., Zhu, Y., Zhang, L. L., Xu, W., Guan, X., Zhang, B., Dong, B., Zhou, X., Zhang, B., et al. rstar2-agent: Agentic reasoning technical report. *arXiv preprint arXiv:2508.20722*, 2025.

Shen, W., Li, C., Chen, H., Yan, M., Quan, X., Chen, H., Zhang, J., and Huang, F. Small llms are weak tool learners: A multi-llm agent. In *EMNLP*, 2024.

Shen, Y., Song, K., Tan, X., Li, D., Lu, W., and Zhuang, Y. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. In *NeurIPS*, 2023.

Sheng, G., Zhang, C., Ye, Z., Wu, X., Zhang, W., Zhang, R., Peng, Y., Lin, H., and Wu, C. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K. R., and Yao, S. Reflexion: language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

Snell, C., Lee, J., Xu, K., and Kumar, A. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

Subramaniam, V., Du, Y., Tenenbaum, J. B., Torralba, A., Li, S., and Mordatch, I. Multiagent finetuning: Self improvement with diverse reasoning chains. *arXiv preprint arXiv:2501.05707*, 2025.

Surís, D., Menon, S., and Vondrick, C. Vipergpt: Visual inference via python execution for reasoning. In *ICCV*, 2023.

Taubenfeld, A., Sheffer, T., Ofek, E., Feder, A., Goldstein, A., Gekhman, Z., and Yona, G. Confidence improves self-consistency in llms. *arXiv preprint arXiv:2502.06233*, 2025.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., et al. Scipy 1.0: Fundamental algorithms for scientific computing in python. *Nature Methods*, 2020.

Wang, G., Li, J., Sun, Y., Chen, X., Liu, C., Wu, Y., Lu, M., Song, S., and Yadkori, Y. A. Hierarchical reasoning model. *arXiv preprint arXiv:2506.21734*, 2025.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models. In *ICLR*, 2022.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *COLM*, 2024.

Xue, Z., Zheng, L., Liu, Q., Li, Y., Zheng, X., Ma, Z., and An, B. Simpletir: End-to-end reinforcement learning for multi-turn tool-integrated reasoning. *arXiv preprint arXiv:2509.02479*, 2025.

Yang, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Li, C., Liu, D., Huang, F., Wei, H., et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024a.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., Huang, C., Lv, C., Zheng, C., Liu, D., Zhou, F., Huang, F., et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q. V., Zhou, D., and Chen, X. Large language models as optimizers. In *ICLR*, 2024b.

Yang, H., Hu, Y., Kang, S., Lin, Z., and Zhang, M. Number cookbook: Number understanding of language models and how to improve it. *arXiv preprint arXiv:2411.03766*, 2024c.

Yoran, O., Wolfson, T., Bogin, B., Katz, U., Deutch, D., and Berant, J. Answering questions by meta-reasoning over multiple chains of thought. In *EMNLP*, 2023.

Yuksekgonul, M., Bianchi, F., Boen, J., Liu, S., Lu, P., Huang, Z., Guestrin, C., and Zou, J. Optimizing generative ai by backpropagating language model feedback. *Nature*, 2025.

Zeng, A., Liu, M., Lu, R., Wang, B., Liu, X., Dong, Y., and Tang, J. Agenttuning: Enabling generalized agent abilities for llms. *arXiv preprint arXiv:2310.12823*, 2023.

Zhang, D., Huang, X., Zhou, D., Li, Y., and Ouyang, W. Accessing gpt-4 level mathematical olympiad solutions via monte carlo tree self-refine with llama-3 8b. *arXiv preprint arXiv:2406.07394*, 2024.

Zhang, D., Wu, J., Lei, J., Che, T., Li, J., Xie, T., Huang, X., Zhang, S., Pavone, M., Li, Y., et al. Llama-berry: Pairwise optimization for olympiad-level mathematical reasoning via o1-like monte carlo tree search. In *NAACL*, 2025.

Zheng, L., Yin, L., Xie, Z., Sun, C. L., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., et al. Sglang: Efficient execution of structured language model programs. In *NeurIPS*, 2024.

Zhou, H., Chen, Y., Guo, S., Yan, X., Lee, K. H., Wang, Z., Lee, K. Y., Zhang, G., Shao, K., Yang, L., et al. Agentfly: Fine-tuning llm agents without fine-tuning llms. *arXiv preprint arXiv:2508.16153*, 2025.

Appendix

A Implementation Details of AlphaApollo

In this section, we provide the employed prompts in the AlphaApollo, including:

- the system prompt of the main model (Figure 9);
- the descriptions for the computational module (Figure 10) and the retrieval module (Figure 11);
- the system prompts for the query rewriter (Figure 12) and result summarizer (Figure 13) in the retrieval module.

A.1 AlphaApollo System Prompt

System prompt of the main model

You should solve the given math problem step by step.
There are some tools that you can use to help you solve the math problem.
Examine the available tools and determine which ones might relevant and useful for addressing the query. Make sure to consider the tool metadata for each tool.
Based on your thorough analysis, decide if your memory is complete and accurate enough to generate the final output, or if additional tool usage is necessary.
Please reason step by step, and put your final answer within \\boxed{}.

Figure 9: **System prompt**. In system prompt, we instruct the model to solve math problems step by step, inspect tool metadata to decide which tools to use, and present the final answer in box.

A.2 Tool Descriptions

Tool description of the computational module

A tool to execute python code. You need to use print() to get the result of the code execution. If you are not familiar with the package (sympy, scipy, numpy, math, cmath, fractions, itertools) you are using, you can call ‘rag_system_retrieve’ tool before calling ‘python_code_executor’ tool.

Args:
    code: code text

Returns:
    The result of the code execution

metadata:
    “limitations”: 1) “Do not perform plotting operations, such as using matplotlib.”, 2) “If you want to use external packages, you can only use sympy, scipy, numpy, math, cmath, fractions, and itertools.”, 3) “Not applicable to geometry and number theory problems.”, 4) “No access to any system resources, file operations, or network requests.”, 5) “All calculations must be self-contained within a single function or script.”, 6) “Input must be provided directly in the query string.”, 7) “Output is limited to numerical results or simple lists/tuples of numbers.”.
    “best_practices”: 1) “Provide clear and specific queries that describe the desired mathematical calculation.”, 2) “Include all necessary numerical inputs directly in the query string.”, 3) “Ensure all required numerical data is included in the query.”.

Figure 10: **Tool description of the computational module**. In this tool description, we instruct the model on how to use computational module.

Tool description of the retrieval module

A search tool system for querying Python package documentation.
Args:
    repo_name: One of [sympy, scipy, numpy, math, cmath, fractions, itertools]
    query: Natural-language query for retrieval, e.g., ‘Function interface and examples of calling sympy to solve nonlinear equations’
    top_k: Number of documents to return per sub-query (default: 3)

Returns:
    str: The query result, including relevant code examples, descriptions, and usage.

metadata:
    “limitations”: 1) “Not suitable for solving specific math problems.”
    “best_practices”: 1) “Ask for usage instructions or examples for a specific function interface whenever possible.”

Figure 11: **Tool description of the retrieval module**. In this tool description, we instruct the model on how to use retrieval module.

A.3 Prompts of the Retrieval Module

System prompt of the query rewriter

You decide whether to generalize the user’s query. If it is already short, clear and general, return an empty line. Otherwise, rewrite it into ONE short, general question about the relevant functions/classes/APIs for this task type. The rewrite should:
(1) Keep only: task category and the library/tool name if present.
(2) Do NOT include any specific objects/structures, numbers, variable names, operators, or concrete equation forms.
(3) Return a single concise question text (no preamble), or empty if the original is already short, clear, and general.
Return ONLY the question text or empty.

Figure 12: **System prompt of the query rewriter**. In the query rewriter prompt, the model is asked to decide whether and how to generalize the user’s query.

System prompt of the result summarizer

You decide whether to answer. Treat the provided docs as optional hints; they may be partial or off-topic. If the docs already clearly answer the query, return an empty line. Otherwise, write a concise, self-contained answer that:
(1) includes fully-qualified function or class names when possible (e.g., sympy.core.function.diff);
(2) includes brief, runnable usage examples;
(3) NEVER mentions files, tests, sources, or ’documents’, and avoids any meta commentary about what the docs do or do not contain;
(4) may rely on general knowledge beyond the docs to help the user.
Return ONLY the answer text (no preamble) or empty.

Figure 13: **System prompt of the result summarizer**. In the result summarizer prompt, the model is asked to decide whether and how to summarize the provided docs.

B Case Studies

B.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 Section 2.2, 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`(Fig. 14);
  - `IndentationError` (Fig. 15);
- Solved with the model-based error correction component.
  - `SyntaxError` (Fig. 16);
  - `NameError` (Fig. 17);
  - `IndexError` (Fig. 18);
  - `TypeError` (Fig. 19);
  - `ValueError` (Fig. 20);
  - `ImportError` (Fig. 21);
  - `AttributeError` (Fig. 22);
  - `NotImplementedError` (Fig. 23).

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 14: 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 15: 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 16: 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 17: 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 18: 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 19: 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 20: 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 21: 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 22: 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 23: The example of `NotImplementedError` in model-generated code scripts.

B.2 Cognitive Behaviors

In this section, we provide the complete model responses for the case studies in Section 3.

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 24: 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 25: 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 $\log_x(y^x) = \log_y\left(x^{4y}\right) = 10$. 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 26: 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:

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

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

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 27: The complete model response for the Backtracking case study.