TIR-Bench: A Better Benchmark for Fine-grained Evaluation of Tool-integrated Reasoning

Anonymous ACL submission

Abstract

We introduce TIR-Bench, a benchmark designed for evaluating Tool-integrated Reasoning (TIR) in large reasoning models (LRMs). TIR-Bench addresses the limitations of existing TIR evaluations, such as narrow task coverage and a lack of fine-grained analysis. Our benchmark spans various domains, including number theory, cryptography, and neuro-symbolic tasks, requiring tool usage for all tasks to effectively decouple intrinsic reasoning abilities from TIR capabilities. It also incorporates automated failure mode analysis, offering insights into model performance. We employ an automated, bottom-up pipeline to generate complex tasks composed of atomic tasks represented in Directed Acyclic Graphs (DAGs). The results of evaluations on multiple LLMs highlight the varying TIR capabilities across models.

1 Introduction

With the rapid development of large reasoning models (LRMs) (Guo et al., 2025; Jaech et al., 2024; Yang et al., 2025a; Meta, 2025), researchers have increasingly realized the importance of Test-time Scaling for enhancing model capabilities (Zhang et al., 2025; Snell et al., 2024; Yang et al., 2025b). However, even the most powerful models struggle to solve certain tasks relying solely on internal reasoning, especially those involving complex calculations and neuro-symbolic tasks. Consequently, researchers have begun to focus on utilizing Tool-integrated Reasoning (TIR) to break through the boundaries of model capabilities (Lin and Xu, 2025).

As the paradigm of using Reinforcement Learning (RL) to train implicit reasoning evolves (Xue et al., 2025; Wang et al., 2025b; Liu et al., 2025; Jin et al., 2025), training models with TIR capability has garnered widespread attention in both academia and industry. However, the vast majority of existing research (Li et al., 2025b; Feng et al., 2025; Bai et al., 2025; Chang et al., 2025; Wang et al., 2025a; Li et al., 2025a; Lin and Xu, 2025) evaluates TIR primarily on mathematical benchmarks such as AIME (MAA, 2024, 2025) and MATH (Hendrycks et al., 2021). These evaluations suffer from several limitations. First, they have limited capability coverage, as experiments are confined to mathematical benchmarks where tool usage is almost exclusively for numerical calculations. Second, there is a lack of tool-using necessity, since not all problems strictly require tools; particularly for larger models whose internal computation is often sufficient[1], making it difficult to decouple intrinsic calculation ability from TIR capability. Finally, they lack fine-grained analysis, typically providing only final results without automated means to analyze process metrics or attribute failure modes.

To address these issues, we propose TIR-Bench, a benchmark specifically designed for TIR. It features broader capability coverage, including not only numerical calculations but also number theory, cryptography, and neuro-symbolic capabilities such as encoding/decoding, date operations, and string manipulations. It ensures the necessity of TIR, as all tasks are impossible to solve without tools, effectively decoupling intrinsic capabilities from tool usage. Furthermore, it enables fine-grained analysis through automated evaluation methods that cover both process metrics and failure mode attribution.

We employ a fully automated bottom-up pipeline to construct TIR-Bench, which is shown in Figure 1. Our motivation is to synthesize complex tasks by organically composing atomic tasks that represent TIR capabilities. Solving these complex tasks necessitates multi-turn, interleaved reasoning and tool usage.

Our analysis reveals a critical dissociation be-

[1]The strongest models like GPT-5.2-thinking achieves 100% accuracy on AIME without tool use.

tween intrinsic reasoning capabilities and TIR performance. We observe that some reasoning models, such as Qwen3-thinking series, do not naturally excel in TIR benchmarks. Instead, these models often exhibit an "overconfidence trap," failing to invoke necessary tools for complex calculations or decryption tasks despite possessing high latent capacity. Furthermore, our analysis indicates that while smaller models share comparable sub-task solving ability with larger counterparts, they lack the planning and compositional abilities required to bridge the gap between potential and actual accuracy.

The contributions of this paper are as follows:

- We propose TIR-Bench, a benchmark constructed specifically for TIR evaluation. It covers a wide range of capabilities, decouples the intrinsic computation ability of the model from its TIR capability, and enables automated fine-grained failure mode analysis to effectively guide model optimization.
- We conduct fine-grained evaluations and analyses of multiple mainstream LLMs, highlighting their strengths and weaknesses in long-horizon TIR, thereby deepening the understanding of current models' TIR capabilities.
- The data construction pipeline is fully automated with no human intervention required. This allows us to scale up the difficulty level for more powerful future models, as well as dynamically update benchmark datasets to mitigate potential data leakage risks.

2 Related Works

2.1 Cross-domain TIR Evaluation

Some works have explored whether TIR can generalize across domains. Chen et al. (2025b) argue that the ability to use tools is domain-independent and demonstrate that models trained on mathematical tasks with TIR can generalize to physics, biology, finance, and philosophy. Similarly, Chen et al. (2025a) address the limitation of single-domain RL by constructing a dataset from SymBench, BBH, and Reasoning-Gym to perform multi-stage RL, showing improvements across multiple domains. However, these works primarily switch evaluation sets from different domains without constructing a dedicated benchmark for TIR, and they still face the issue of being unable to decouple intrinsic calculation ability from tool usage capability.

Benchmark	# Tasks	Multiple Capabilities	Tool Necessity	Fine-grained Analysis
AIME	60	×	×	×
CARP	976	×	✓	×
ReasonZoo	3015	✓	×	×
TIR-Bench	510	✓	✓	✓

Table 1: Comparison of Benchmarks for TIR Evaluation.

2.2 TIR Benchmark

Other works consider building benchmarks specifically for TIR but have certain limitations. Zhang et al. (2023) point out that existing tool-augmented research often requires only a single tool call. They propose the CARP dataset, ensuring through a rigorous "automatic filtering + manual verification" process that problems require multi-step calculation and reasoning. However, CARP primarily focuses on numerical calculation capabilities, and the reasoning steps are relatively few (average 4.7 steps). Zhao et al. (2025) propose REASONZOO, a benchmark covering nine major categories to evaluate TIR. While their motivation shares similarities with ours, there are distinct differences. First, they largely reuse or simply adapt existing benchmarks and public questions, whereas our questions are original. Second, their benchmark focuses on shorter reasoning processes with fewer steps, while ours involves extremely long, multi-step reasoning. Finally, many problems in their benchmark can be solved by models directly without tools, failing to decouple intrinsic calculation capability from tool usage capability.

Crucially, none of these works provide fine-grained attribution analysis for failures in long-horizon reasoning process to guide model optimization, a gap that our work fills. The comparison of TIR-Benchand other benchmarks for TIR evaluation is outlined in Table 1.

3 Benchmark Construction

We adopt a fully automated, bottom-up benchmark construction pipeline, which is shown in 1. Starting from atomic tasks where each represents a unique capability, we combine them according to a given Directed Acyclic Graph (DAG) representing abstract solution steps to form a complex problem. This problem must be solved by the model through interleaved tool calls during the thinking process.

3.1 DAG Generation

We generate DAGs to represent the logical structure of complex problems. We specify the number

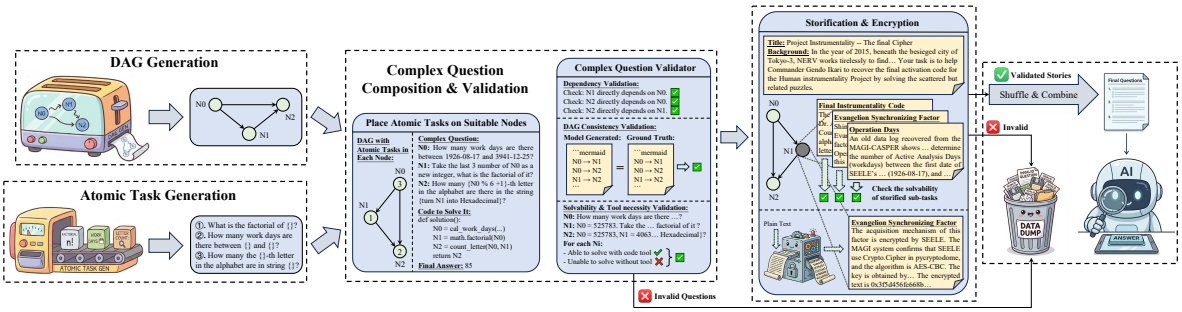


Figure 1: The overall data construction pipeline of our TIR-Bench. We start from generating abstract DAGs to represent the process of problem solving, then combining the atomic tasks according to the DAGs to form complex questions. We then validate the complex questions with a rigorous three-stage process, and finally storify the questions and encrypt sub-tasks to form the final questions that require interleaved tool calls during the thinking process to solve.

Difficulty	Nodes (n)	Edges (m)	Count	Total
Easy	3–5	2–7	4/config	24
Medium	6–8	8–13	4/config	32
Hard	9–11	11–17	4/config	32
Extreme	14–16	20–28	3/config	24

Table 2: Statistics of generated DAGs across different difficulty levels.

Category	Sub-categories
Big Integer Calculation	11
Encoding/Decoding	3
Number Theory	12
Cryptography	9
Date Operations	4
String Operations	9
Total	**48**

Table 3: Statistics of Atomic Tasks.

of nodes n and edges m, where n atomic tasks correspond to n nodes, and m edges correspond to the dependencies between solution steps. To ensure the DAGs represent realistic puzzle-solving strategies—neither too complex nor too simple—each node has a maximum in-degree of 3, which means the corresponding sub-task depends on 3 input variables, and a maximum out-degree of 2, which means the answer of the corresponding sub-task is the input variable of at most 2 different sub-tasks.

We control the difficulty through the number of nodes and edges, dividing the benchmark into four difficulty levels: Easy, Medium, Hard, and Extreme. The statistics of the generated DAGs are shown in Table 2. For more details about the generated DAGs, please refer to Appendix B

3.2 Atomic Task Generation and Validation

We define six major categories of tasks that models cannot complete without tools: Big Integer Calculation, Number Theory, Cryptography, and neuro-symbolic tasks like Date Operations, Encoding/Decoding, and String Operations. These are further divided into 48 sub-categories, whose distribution is shown in Table 3. Each sub-category has a specific schema template and the corresponding ground truth code to solve it. The schema templates have placeholders for the input values of each atomic task, where the input values is also the input variables of the corresponding ground truth code. You can refer to the left-bottom part of Figure 1 for examples. Adhering to validity constraints (e.g., the divisor can not be zero), we randomly generate input values and tailored heuristics for every atomic task to ensure the complexity of the generated question exceeds the LLMs' internal capacity for solving.

To ensure the necessity of tools, we validate each atomic task. A task is considered valid only if the state-of-the-art LLMs fail to solve it 10 times without a code interpreter tool but succeed at least once out of 10 attempts with a code interpreter tool. In our experiments, we adopt GPT-5.2 and Gemini-3-pro as the validators for atomic tasks.

3.3 Complex Question Composition

We combine a DAG and a set of randomly sampled atomic tasks into a complex problem using Gemini-2.5-pro. The model is prompted to organically combine these elements into a complex question that conforms to the DAG's solution path. The generation process follows a topological order, where the model selects atomic tasks as nodes, writes code to calculate their output values, and selects

Algorithm 1 Complex Question Composition

Require: DAG $G = (V, E)$, Set of Atomic Tasks $\mathcal{T}$
Ensure: Complex Question Q, Solution Code C, Final Answer A

Initialize empty map M for node-task assignments
Initialize empty list L for code lines
for each node v in topological order of G **do**
  if v is a source node (in-degree 0) **then**
    Select an atomic task $t \in \mathcal{T}$
    Generate code c_v to solve t and get answer a_v
  else
    Identify predecessor nodes $P_v = \{u | (u, v) \in E\}$
    Retrieve answers $\{a_u | u \in P_v\}$
    Select an atomic task $t \in \mathcal{T}$ suitable for inputs $\{a_u\}$
    Modify $\{a_u\}$ accordingly to ensure t requires complex computation (e.g., large numbers)
    Generate code c_v to solve t using $\{a_u\}$ as inputs
  end if
  Store assignment $M[v] \leftarrow (t, c_v, a_v)$
  Append c_v to L
end for
Compose question text Q from $\{M[v].t\}$ descriptions
Combine L into complete solution code C
$A \leftarrow$ answer of the sink node
return Q, C, A

the next node based on these values, ensuring input values are reasonable. The prompt used for this process is detailed in Appendix A.1. The composition algorithm is outlined in Algorithm 1.

The algorithm proceeds as follows: First, we initialize a map to store task assignments for each node and a list to collect the generated code. We then iterate through each node in the DAG according to its topological order. If a node is a source node (in-degree 0), we select an appropriate atomic task and generate code to solve it, obtaining its answer. If a node is not a source node, we identify its predecessors and retrieve their answers. We then select an atomic task suitable for these inputs, modifying the inputs if necessary to ensure the task requires complex computation (e.g., involving large numbers). We generate code to solve this task using the predecessor answers as inputs. Finally, we compose the question text from the descriptions of all assigned tasks and combine the code snippets into a complete solution script, with the answer of the sink node serving as the final answer.

3.4 Validation of Complex Questions

The generated complex questions undergo a rigorous three-step validation process:

Dependency Verification We verify that every variable truly depends on its direct predecessors. We parse the "code" field output by the model using Abstract Syntax Tree (AST) to identify assignment statements. We then randomly perturb the value of the direct predecessor variables 10 times. If the variable's value changes at least 5 times, the dependency is confirmed. It is worth noting that we exclude all indirect dependencies to ensure that only direct dependencies are detected. For example, in the case of $N_0 \to N_1 \to N_2$ and $N_0 \to N_2$, the connection $N_0 \to N_2$ involves both direct and indirect paths. Under this circumstance, when studying the direct dependency of N_0 to N_2, we modify the predecessor (N_0) after the intermediate node (N_1) is assigned, ensuring only the direct dependency is detected.

DAG Consistency Verification We verify that the generated question matches the original DAG structure. We prompt an LLM to reconstruct the DAG structure from the generated question in Mermaid format. An example of the Mermaid format is provided in Appendix A.3. Comparing this reconstructed DAG with the ground truth DAG ensures consistency. Since the raw questions follow a fixed format, generating the DAG Mermaid from the question is straightforward. We compiled a test set of 100 manually verified samples, and Gemini-2.5-pro achieved a 100% accuracy in Mermaid generation. Therefore, this verification method is reliable. The prompt for DAG reconstruction is in Appendix A.2.

Solvability and Tool Necessity Verification We decompose the complex problem into sub-tasks. For each sub-task, given the correct values of its predecessor tasks as the input values, we construct a self-contained sub-problem. A sub-task is considered as tool-necessary if Gemini-2.5-pro fails to solve its corresponding sub-problem directly without tools (0 successes in 5 attempts), and is considered as solvable if Gemini-2.5-pro succeeds to solve it with a code interpreter tool (at least 1 success in 10 attempts). A complex problem is valid if all its sub-tasks are solvable, and at least a proportion α of sub-tasks are tool-necessary. In our experiments, α is set to 0.8. The prompts for this verification are in Appendix A.4.

3.5 Storification and Encryption

To further enhance the benchmark, we wrap the validated questions in stories based on movies or literature. The model rewrites the problem description, embedding each sub-task into the story context. We verify the correctness of these storified sub-tasks by checking if they can be solved by strong LLMs given the correct predecessor values.

We also introduce encryption to selected subtask nodes. This serves two primary purposes: First, it tests the model's cryptography knowledge and its ability to use tools for decryption. Second,

and more importantly, it forces a "stop and think" mechanism. Without encryption, a model might theoretically solve the entire chain of sub-tasks in a single pass if it understands all dependencies. By encrypting the task description using one of 9 algorithms (e.g., AES, DES, ChaCha20) with keys derived from the outputs of predecessor nodes, we ensure the model must pause to decrypt the description before it can understand the task and proceed. This enforces a multi-step, interleaved process of "reasoning -> tool use -> result acquisition -> continue reasoning", which is the essential process of TIR.

Specifically, we randomly select 30% nodes for encryption. For a selected node, we generate a key from the outputs of its predecessor nodes, then encrypt the task description of the node using a randomly selected encryption algorithm. The lower-bottom part of Figure 1 illustrates an example of the encrypted node, where the encrypted description, along with the algorithm name and the key derivation rule, replaces the original description of the selected node.

3.6 Final Assembly

Finally, we assemble the storified sub-tasks into a complete problem. Instead of presenting them in topological order, we randomly shuffle the sub-tasks. This forces the model to reason about dependencies and reconstruct the solution DAG, significantly increasing the difficulty. The prompt for the final question assembly is provided in Appendix A.5.

4 Experiments

This section evaluates tool-integrated reasoning (TIR) on TIR-BENCH across (i) frontier closed-source models and (ii) mainstream open-source models, covering both dedicated reasoning models and general-purpose chat models. We report end-to-end accuracy across difficulty levels to provide a comprehensive overview of current models' TIR capabilities.

4.1 Experimental Settings

Models. We evaluate a wide range of models, including frontier closed-source models such as GPT-5, and GROK-4, as well as mainstream open-source models like the QWEN3 and DEEPSEEK families.

Evaluation Modes. Models vary in their support for tool integration. Some frontier models provide native support for code interpretation via their proprietary APIs, allowing them to manage tool calls and execution internally. For these models, we utilize the **NATIVE** mode, where the model is given the question and uses its native TIR capability to solve it. Other models, which lack such native capabilities, must rely on an external code execution environment. We evaluate these models using the **MULTI** mode, where tool usage is implemented as a controlled multi-turn protocol: we write prompts to instruct the model, so that whenever code is needed, the model outputs a Python program, we execute it in a restricted sandbox, and return the stdout and error messages as the next-turn observation. The **MULTI** mode enables the evaluation of a model's TIR potential, even if the model does not natively support TIR. This provides valuable guidance for model selection prior to Reinforcement Learning (RL), as combining Chat models with multi-turn tool-use prompts is often the starting point for RL (Li et al., 2025b).

Difficulty levels. We report results on four difficulty levels: Easy, Medium, Hard, and Extreme, which are determined by the complexity of the underlying DAG (cf. Table 2).

Decoding and API setup. Unless otherwise specified, we use each provider's default decoding settings and evaluate all models via a unified API framework. Full configurations and execution details are provided in Appendix C.

4.2 Metrics

End-to-end accuracy. For each subset $\mathcal{S}$ of instances, accuracy is:

$$\mathrm{Acc}(\mathcal{S}) = \frac{1}{|\mathcal{S}|} \sum_{x \in \mathcal{S}} \mathbb{I}\big[\hat{y}(x) = y(x)\big], \quad (1)$$

where $y(x)$ is the ground-truth final answer and $\hat{y}(x)$ is the model output parsed from `\boxed{}`.

Node-level correctness (used in analyses). Each instance x contains a DAG with node set $V(x)$ and ground-truth intermediate values $\{a_v(x)\}_{v \in V(x)}$. We require the model to output intermediate node values $\{\hat{a}_v(x)\}$ as a JSON object (Appendix A.5). We define node accuracy as:

$$\mathrm{NA}(x) = \frac{1}{|V(x)|} \sum_{v \in V(x)} \mathbb{I}\big[\hat{a}_v(x) = a_v(x)\big]. \quad (2)$$

where missing or unparsable $\hat{a}_v(x)$ is treated as incorrect.

Model	Easy		Medium		Hard		Extreme	
	Acc	Sub	Acc	Sub	Acc	Sub	Acc	Sub
NATIVE *Mode*								
GPT-5	**92.7**	**85.4**	**88.0**	**85.9**	**82.9**	**84.7**	**76.9**	**80.6**
QWEN3-MAX-PREVIEW	85.5	82.8	67.4	68.9	47.9	65.3	48.1	59.1
GROK-4	70.9	71.2	53.3	56.2	23.1	27.5	3.8	6.6
MULTI *Mode*								
DEEPSEEK-REASONER-3.2	**94.5**	**91.0**	**87.0**	**85.1**	**78.6**	**80.0**	48.1	47.9
GEMINI-3-PRO-PREVIEW	81.8	79.0	69.6	72.0	56.4	67.0	**59.6**	**64.1**
DEEPSEEK-CHAT-3.2	89.1	89.1	77.2	74.7	49.6	56.6	32.7	38.4
GEMINI-2.5-PRO	43.6	59.7	39.1	59.8	41.9	65.0	28.8	53.4
GPT-5	60.0	69.5	44.6	65.1	22.2	46.8	11.5	45.3
GEMINI-2.5-FLASH	49.1	65.7	32.6	52.2	27.4	52.2	13.5	42.3
QWEN3-MAX-PREVIEW	63.6	70.8	40.2	59.6	29.9	59.0	25.0	46.7
QWEN3-235B-A22B-INSTRUCT	70.9	77.6	42.4	59.8	29.1	51.6	13.5	34.0
QWEN3-235B-A22B-THINKING	56.4	50.6	31.5	42.7	15.4	27.4	9.6	18.9
QWEN3-NEXT-80B-A3B-INSTRUCT	21.8	37.8	9.8	26.0	3.4	13.0	0.0	12.8
QWEN3-NEXT-80B-A3B-THINKING	34.5	23.2	13.0	11.2	6.0	9.9	3.8	7.4

Table 4: End-to-end accuracy (Acc) and Subtask completion rate (Sub) in percentage (%) on TIR-BENCH under NATIVE and MULTI evaluation modes. The best result in each column (within each mode) is in bold, and the second best is underlined.

4.3 Main Results

We report end-to-end accuracy by difficulty level for both NATIVE and MULTI evaluation modes.

4.3.1 Accuracy Analysis

In the NATIVE mode, GPT-5 demonstrates a commanding lead, significantly outperforming the runner-up QWEN3-MAX-PREVIEW, particularly on the Extreme subset (76.9% vs. 48.1%), while GROK-4 shows limited capability. In the MULTI mode, DEEPSEEK-REASONER-3.2 achieves the best overall performance, indicating strong native TIR potential. However, GEMINI-3-PRO-PREVIEW surpasses it by 11.5% on the Extreme subset (59.6% vs. 48.1%), highlighting superior long-chain reasoning capabilities. Within the QWEN3 series, closed-source and larger models consistently dominate. Notably, *Thinking* variants underperform their *Instruct* counterparts (e.g., 9.6% vs. 13.5% on Extreme). As indicated in Figure 2, this is due to the overconfidence of *Thinking* models, which tend to rely on internal reasoning rather than tool invocation, suggesting they are less suitable as initialization checkpoints for TIR training.

4.3.2 Impact of Evaluation Modes

Top-performing models like GPT-5 and QWEN3-MAX-PREVIEW exhibit a substantial performance gap between NATIVE and MULTI modes. For instance, GPT-5 sees a 65.4% accuracy gain on the Extreme subset in NATIVE mode. This disparity serves as a quantitative indicator of the efficacy of their TIR-specific training, validating their successful adaptation to TIR tasks.

5 Fine-grained Analysis

In Section 4, we reported the end-to-end accuracy of various models across different difficulty levels. To gain a deeper understanding of the failure modes and the capabilities of current models in tool-integrated reasoning (TIR), we conduct a fine-grained analysis in this section. This analysis consists of three parts: (1) **Failure Modes** (Section 5.1), which identifies the root causes of failures; (2) **Reasoning Gap** (Section 5.2), which quantifies the gap between the theoretical upper bound calculated from sub-task accuracies and the observed model accuracy, and evaluates the model's capability for composing the sub-task solving ability to the long-horizon reasoning (Lu et al., 2025); and (3) **Atomic Capability Loss** (Section **??**), which measures the extent of atomic capability degradation in long-horizon reasoning. Note that failure modes analysis is a full automated and standard part of our evaluation, which requires no additional model inference process. Therefore, it is conducted on all evaluated model. As for the reasoning gap and the atomic capability loss, our analysis focuses primarily on the Qwen3 family(Yang

Type	Description
Transcription Error	Internal tool result correct, model miscopies it to the output.
Calculation Error	Model writes code to do tasks, but code logic yields wrong results.
Decryption Logic Error	Input parameters are correct, but the decryption method is wrong.
Ciphertext Copy Error	The ciphertext is miscopied from the question to the code.
Give-up (Null)	The model concludes that it is unable to solve the task.
No Tool / Hallucination	Model does tasks without tools, or simply hallucinates the answer.

Table 5: Failure Modes Taxonomy.

et al., 2025a), covering a range of model sizes. As they are the most popular open-source models, our findings can provide valuable insights to the research community.

5.1 Failure Modes

Since our evaluation tasks possess a topological structure where errors in upstream sub-tasks inevitably lead to failures in downstream ones, localizing the first incorrect sub-task and attributing its failure modes is crucial for understanding model limitations. The bottom-up data construction approach renders our benchmark inherently amenable to automated failure localization and attribution. Therefore, we localize the first failure along the ground-truth problem-solving topological order of sub-tasks, and attribute its failure mode using a trace-based, rule-driven analyzer.

Failure Localization. Each instance x is associated with a ground-truth DAG whose nodes are topologically ordered as $\pi(x) = (v_1, \ldots, v_{|V(x)|})$. Since we require the model to output intermediate node values $\{\hat{a}_v(x)\}$ in a JSON object, we scan nodes in topological order and compare the node value with the ground truth value $a_{v_k}(x)$ to identify the smallest index k such that $\hat{a}_{v_k}(x) \neq a_{v_k}(x)$, and denote v_k as the *failure node*.

Failure-mode Taxonomy. We categorize the cause of the failure node into six mutually exclusive types, whose elaborations are listed in Table 5. Operational definitions and the full attribution rules are provided in Appendix C.3.

Results and Analysis Figure 2 illustrates the distribution of error attribution types across different models under NATIVE and MULTI evaluation modes. The analysis reveals distinct failure patterns characteristic of specific model families.

In the NATIVE setting, GPT-5 and QWEN3-MAX-PREVIEW exhibit "No Tool / Hallucination" as the predominant failure mode, suggesting an over-reliance on internal computation capabilities. While QWEN3-MAX-PREVIEW displays minor incidences of calculation and transcription errors, these are secondary to its hallucination issues. Conversely, GROK-4 primarily suffers from "Calculation Error" and "Refusal (Null)," indicating a necessity to enhance its capability in leveraging code-based tools for problem-solving.

Under the MULTI mode, the DEEPSEEK series and GPT-5 show frequent "Decryption Logic Errors," pointing to potential deficiencies in domain-specific cryptographic knowledge. Notably, DEEPSEEK-REASONER-3.2 incurs a significant portion of "Calculation Errors"; despite its strong intrinsic reasoning faculties, this highlights limitations in effectively utilizing code interpreters. The GEMINI series demonstrates a tendency towards "No Tool / Hallucination"—implying overconfidence—and struggles with "Ciphertext Copy Errors" during long-context transcription, although GEMINI-3-PRO-PREVIEW shows marked improvement over the v2.5 iterations. Finally, the open-source QWEN3 models are dominated by "No Tool / Hallucination" errors, underscoring a critical need to reinforce tool-invocation behaviors in appropriate contexts.

This heterogeneity in error distribution confirms that failure modes are highly model-dependent. Consequently, our automated failure mode analysis serves as a vital diagnostic mechanism, providing actionable insights for targeted model optimization.

5.2 Reasoning Gap

We next quantify the performance loss specifically attributable to the model's inability to effectively compose sub-task solving ability into the long-horizon reasoning process. Even if a model can solve every individual sub-task correctly when presented in isolation, it may still fail the end-to-end problem due to deficiencies in planning, state tracking, or error propagation management. We term this deficit the ***Reasoning Gap***. It represents the difference between a theoretical performance upper bound (assuming oracle-level composition of sub-tasks) and the actual observed accuracy.

Isolated sub-task evaluation. Each benchmark instance x contains node-level sub-tasks $\{v \in V(x)\}$. For each node v, we evaluate the model on the corresponding sub-task with all predecessor values provided as ground truth. We repeat this isolated evaluation 5 times and compute the empirical

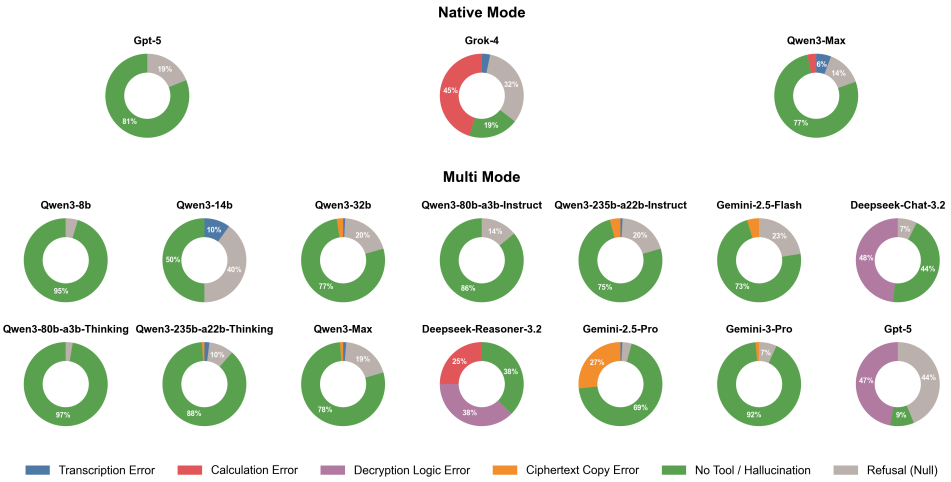


Figure 2: Distribution of error attribution types for all models under NATIVE and MULTI evaluation modes. The charts aggregate errors across all difficulty levels for each model.

success rate:

$$p_{x,v} = \frac{1}{K}\sum_{k=1}^{K}\mathbb{I}\big[\hat{a}_{x,v}^{(k)} = a_{x,v}\big], \quad K = 5. \tag{3}$$

Per-instance theoretical upper bound. Assuming independence across node-level successes, the per-instance upper bound is:

$$U(x) = \prod_{v \in V(x)} p_{x,v}. \tag{4}$$

Dataset-level upper bound and reasoning gap. For a dataset split $\mathcal{S}$ (e.g., all Hard instances), we compute:

$$\begin{aligned} U(\mathcal{S}) &= \frac{1}{|\mathcal{S}|}\sum_{x \in \mathcal{S}} U(x), \\ G(\mathcal{S}) &= U(\mathcal{S}) - \mathrm{Acc}(\mathcal{S}), \end{aligned} \tag{5}$$

where $G(\mathcal{S})$ is the *reasoning gap*: the drop from a per-node solvability upper bound to the actual end-to-end accuracy.

Table 6 illustrates a distinct trend where the reasoning gap narrows as model size increases. Smaller models (e.g., QWEN3-8B) exhibit upper bounds comparable to larger counterparts yet suffer from negligible actual accuracy, resulting in substantial gaps. This discrepancy suggests that while smaller models possess sufficient latent reasoning potential, they require significant optimization to realize it. Furthermore, the data indicates that model scale is pivotal for effective reasoning planning and the ability to compose sub-task solutions into long-horizon reasoning processes. Notably, thinking variants display larger Reasoning Gaps compared to standard instruction-tuned models (e.g., QWEN3-235B-THINKING vs. QWEN3-235B-INSTRUCT). As previously analyzed, this performance drop is attributed to an over-reliance on internal computation; thus, the widened gap reflects a specific deficiency in Tool-Integrated Reasoning (TIR) capabilities rather than a lack of intrinsic reasoning power.

Model	Upper Bound	Actual Acc	Gap
QWEN3-8B	82.6	0.6	82.0
QWEN3-14B	85.8	2.6	83.2
QWEN3-32B	87.9	5.7	82.2
QWEN3-80B-INSTRUCT	77.9	7.9	70.0
QWEN3-80B-THINKING	85.5	12.7	72.8
QWEN3-235B-INSTRUCT	91.3	37.7	53.6
QWEN3-235B-THINKING	92.6	26.3	66.3

Table 6: Reasoning Gap analysis for the QWEN3 family. Upper Bound is the theoretical accuracy assuming independent success of sub-tasks. Gap represents the performance loss due to compositional failures.

6 Conclusion

TIR-Bench offers a comprehensive and scalable framework for assessing the TIR abilities of LLMs, filling gaps in existing benchmarks. By incorporating multi-step reasoning and a wide variety of tool-using tasks, it provides a more accurate measure of models' capabilities in real-world scenarios requiring tool invocation. The fine-grained analysis of failure modes and reasoning gaps will guide the optimization of future models, pushing the boundaries of tool-integrated reasoning. The data construction pipeline is automated, enabling continuous updates and scalability to meet the challenges posed by increasingly advanced models.

Limitations

TIR-Bench currently focuses primarily on evaluating models using a code interpreter, leaving other potential tools and integrations underexplored. Future iterations could incorporate a broader range of tools to assess tool-integrated reasoning more comprehensively.

The current difficulty level of tasks in TIR-Bench may still be insufficient for the most advanced models, such as GPT-5. As these models continue to evolve, there will be a need to increase the complexity of tasks to fully challenge their capabilities.

The atomic tasks in TIR-Bench are relatively simple at this stage. In future versions, we aim to integrate more challenging reasoning tasks, such as those from AIME and other high-difficulty problem sets, to provide a more robust evaluation of models' reasoning abilities.

References

Fei Bai, Yingqian Min, Beichen Zhang, Zhipeng Chen, Wayne Xin Zhao, Lei Fang, Zheng Liu, Zhongyuan Wang, and Ji-Rong Wen. 2025. Towards effective code-integrated reasoning. *arXiv preprint arXiv:2505.24480*.

Qikai Chang, Zhenrong Zhang, Pengfei Hu, Jun Du, Jiefeng Ma, Yicheng Pan, Jianshu Zhang, Quan Liu, and Jianqing Gao. 2025. Thor: Tool-integrated hierarchical optimization via rl for mathematical reasoning. *arXiv preprint arXiv:2509.13761*.

Yongchao Chen, Yueying Liu, Junwei Zhou, Yilun Hao, Jingquan Wang, Yang Zhang, Na Li, and Chuchu Fan. 2025a. R1-code-interpreter: Llms reason with code via supervised and multi-stage reinforcement learning. *Preprint*, arXiv:2505.21668.

Zhengyu Chen, Jinluan Yang, Teng Xiao, Ruochen Zhou, Luan Zhang, Xiangyu Xi, Xiaowei Shi, Wei Wang, and Jinggang Wang. 2025b. Can tool-integrated reinforcement learning generalize across diverse domains? *arXiv preprint arXiv:2510.11184*.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. 2025. Retool: Reinforcement learning for strategic tool use in llms. *arXiv preprint arXiv:2504.11536*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, and 1 others. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720*.

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

Chengpeng Li, Zhengyang Tang, Ziniu Li, Mingfeng Xue, Keqin Bao, Tian Ding, Ruoyu Sun, Benyou Wang, Xiang Wang, Junyang Lin, and 1 others. 2025a. Cort: Code-integrated reasoning within thinking. *arXiv preprint arXiv:2506.09820*.

Xuefeng Li, Haoyang Zou, and Pengfei Liu. 2025b. Torl: Scaling tool-integrated rl. *arXiv preprint arXiv:2503.23383*.

Heng Lin and Zhongwen Xu. 2025. Understanding tool-integrated reasoning. *arXiv preprint arXiv:2508.19201*.

Wenhao Liu, Zhengkang Guo, Mingchen Xie, Jingwen Xu, Zisu Huang, Muzhao Tian, Jianhan Xu, Muling Wu, Xiaohua Wang, Changze Lv, and 1 others. 2025. Recast: Strengthening llms' complex instruction following with constraint-verifiable data. *arXiv preprint arXiv:2505.19030*.

Yi Lu, Jianing Wang, Linsen Guo, Wei He, Hongyin Tang, Tao Gui, Xuanjing Huang, Xuezhi Cao, Wei Wang, and Xunliang Cai. 2025. R-horizon: How far can your large reasoning model really go in breadth and depth? *arXiv preprint arXiv:2510.08189*.

MAA. 2024. Aime 2024.

MAA. 2025. Aime 2025.

AI Meta. 2025. The llama 4 herd: The beginning of a new era of natively multimodal ai innovation. *https://ai. meta. com/blog/llama-4-multimodal-intelligence/, checked on*, 4(7):2025.

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

Haozhe Wang, Long Li, Chao Qu, Weidi Xu, Fengming Zhu, Wei Chu, and Fangzhen Lin. 2025a. To code or not to code? adaptive tool integration for math language models via expectation-maximization. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 3060–3075.

Hongru Wang, Cheng Qian, Wanjun Zhong, Xiusi Chen, Jiahao Qiu, Shijue Huang, Bowen Jin, Mengdi Wang, Kam-Fai Wong, and Heng Ji. 2025b. Otc: Optimal tool calls via reinforcement learning. *arXiv e-prints*, pages arXiv–2504.

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

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025a. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Wenkai Yang, Shuming Ma, Yankai Lin, and Furu Wei. 2025b. Towards thinking-optimal scaling of test-time compute for llm reasoning. *arXiv preprint arXiv:2502.18080*.

Beichen Zhang, Kun Zhou, Xilin Wei, Xin Zhao, Jing Sha, Shijin Wang, and Ji-Rong Wen. 2023. Evaluating and improving tool-augmented computation-intensive math reasoning. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Qiyuan Zhang, Fuyuan Lyu, Zexu Sun, Lei Wang, Weixu Zhang, Wenyue Hua, Haolun Wu, Zhihan Guo, Yufei Wang, Niklas Muennighoff, and 1 others. 2025. A survey on test-time scaling in large language models: What, how, where, and how well? *arXiv preprint arXiv:2503.24235*.

Yufeng Zhao, Junnan Liu, Hongwei Liu, Dongsheng Zhu, Yuan Shen, Songyang Zhang, and Kai Chen. 2025. Dissecting tool-integrated reasoning: An empirical study and analysis. *arXiv preprint arXiv:2508.15754*.

A Prompts

A.1 Complex Question Generation Prompt

The prompt shown in Figure 3 and Figure 4 is used to generate complex questions from a DAG and atomic tasks.

A.2 DAG Reconstruction Prompt

The prompt shown in Figure 5 is used to verify the consistency between the generated question and the original DAG structure.

A.3 Mermaid Example

Figure 6 shows an example of the Mermaid format used to represent the DAG structure.

A.4 Solvability and Tool Necessity Verification Prompts

The prompts shown in Figure 7 and Figure 8 are used to verify if a sub-task is solvable with code and unsolvable without code.

A.5 Final Question Assembly Prompt

The prompt shown in Figure 9 is used to assemble the final question presented to the model.

B DAG Examples

The generated DAG examples are show in Figure 10, where we present 4 difficulty levels. Easy: 5 nodes and 6 edges; Medium: 8 nodes and 13 edges; Hard: 11 nodes and 17 edges; Extreme: 16 nodes and 28 edges.

C Experimental Details

C.1 Decoding and API Setup

We evaluate all models using a unified evaluation framework that standardizes API interactions, tool usage protocols, and parameter configurations. This ensures fair comparisons across different model families and providers.

API Configuration. All models are accessed via their respective official APIs or unified gateways. We use the following configurations:

- **Temperature**: We use the default temperature settings for each model provider (e.g., 1.0 for OpenAI and Anthropic, 0.9 for Google) to balance determinism and creativity appropriate for mathematical reasoning. We do not enforce a uniform temperature across all models to respect their optimal operating conditions.
- **Max Tokens**: For most models, we use the default maximum output token limit. For models requiring extended generation capabilities, we set a limit of 32,768 tokens to manage latency.
- **Timeout**: Timeout thresholds are tailored to model inference speeds, ranging from 180 seconds for faster models (e.g., GEMINI-2.5-FLASH) to 900 seconds for slower or more complex models (e.g., GPT-5 in native mode).
- **Retries**: We implement a robust retry mechanism with exponential backoff. We allow up to 1 retry for timeout errors and up to 4 retries for other API errors (e.g., rate limits, server errors).

Tool Calling Implementation. Our framework supports three types of tool calling mechanisms:

1. **Native Function Calling**: For models supporting OpenAI-compatible tool definitions (e.g., GPT-5, CLAUDE-SONNET-4.5, QWEN3), we pass tool definitions via the API's `tools` parameter and parse structured tool calls from the response.
2. **Prompt-based Tool Calling**: For models without native tool support (e.g., GEMINI-2.5), we use a system prompt to instruct the model to output Python code blocks (wrapped in "‘`python ...` "‘). We then extract and execute these blocks using regular expressions.
3. **Native TIR API**: For models with proprietary tool-integrated reasoning endpoints (e.g., Native mode for GPT-5 and QWEN3), we use their specific single-turn APIs where tool execution is handled internally by the provider.

System Prompts. We use specific system prompts to guide models in the MULTI mode:

- **OpenAI-compatible models**: Instructed to use the provided `code_interpreter` tool, execute one step at a time, and wait for results before proceeding.
- **Text-based models**: Instructed to write independent Python code blocks for calculation and verification, and to wait for execution outputs provided by the user.

All prompts emphasize that code blocks must be self-contained and that the model should not hallucinate execution results.

Prompt for Complex Question Generation (Part 1)

I now have [NUM_TASKS] atomic tasks and their answers, and a Directed Acyclic Graph (DAG) combining atomic tasks (given in mermaid format). Please appropriately modify the parameters and descriptions of the atomic tasks to organically combine them into a large task that conforms to the DAG.
Atomic tasks have the following characteristics:

- They receive several parameters as input and output a result;
- They require writing code to complete; it is very difficult for the model to complete them on its own;

The DAG can be interpreted as follows:

- It represents the dependency relationship between the sub-tasks required to complete the large task;
- The edges between nodes indicate that the result of one task is a condition for another task, for example:
  - N0 –> N5, N3 –> N5, means that the answer to N0 is a condition (input) for the problem of N5, and the answer to N3 is another condition (input) for the problem of N5;
  - N0 –> N2, N0 –> N3, means that the answer to N0 serves as a condition for both N2 and N3;

You must strictly follow the steps below to generate the question:

- First, select suitable atomic tasks for the source nodes with an in-degree of 0, and write code to call the python tool to calculate the answers for the atomic tasks;
- For each node that has been arranged, **based on the content, length, number size, etc. of its answer (output)**, flexibly select a suitable atomic task as the problem for the successor node, requiring:
  - Ensure the difficulty of the successor node's problem, i.e., **must ensure that its calculation volume is very large**, making it difficult for the model to complete without using code tools;
  - Methods that can be adopted include:
    - Large number calculation: Problems involving calculation need to construct huge numbers, such as power operations of 199-299 (both base and exponent are in this range, e.g., 248**287); factorials of 300-500; huge terms of various sequences; the nth prime number (n is very large), etc.;
    - Neuro-symbolic operations: Digit statistics in strings need to construct huge strings, such as a string obtained by [continuously dividing a large number by 26, mapping remainders 0-25 to a-z, until the quotient is 0], etc.;
    - There are other means, you can use your creativity.
  - The connection between the preceding problem and the succeeding problem should be as natural as possible;
  - The answer to the sub-task should be as complex as possible and not easily guessed by educational guess. For example, prime factorization can take a certain prime factor in reverse order as the answer, rather than the first or second one, because they are usually 2 or 3;
- Repeat the above steps, repeating the process of [selecting a sub-problem for the node -> setting the sub-problem -> calling the python tool to get the node answer -> re-selecting if the difficulty is not high enough -> jumping to the successor node], until the sink node is arranged with a problem and the final output is obtained;
- After arranging the problems for all nodes, combine the above into a whole question. The question needs to follow the following format, similar to: N0: XXX. N1: XXX. N2: Taking N0 and N1 as input, XXX. N3: Taking the last four digits of N0 and N2 as input, XXX.N4: Taking the years of N1 and N2 as input, XXX.
- At the same time, the solution method for the entire question needs to be written as a complete piece of code, requiring that the answer value of the final node can be obtained eventually:
  - In the output code field, the code must use variable names in the form of N0, N1, N2 for each node, i.e., Nx represents the output value of node x;
  - In the output code field, the calculation of node values must be performed in the order of N0, N1, N2...N_n, rather than the topological order of the DAG;
- Finally, present the question, the complete code, and the answer to the final node of the question in a code block in json format to me, as follows:

```json
{
    "question": "xxx",
    "code": "xxx",
    "answer": "xxx"
}
```

Figure 3: Prompt for Complex Question Generation (Part 1)

Prompt for Complex Question Generation (Part 2)

You need to follow the following requirements:

- Reasonably select each task and place it on a reasonable node;
- According to the content of the DAG, use the output of some tasks as the input of other tasks. Note that the DAG must be correctly reflected, and existing input conditions cannot be omitted or non-existent ones added;
- The final node needs to output a result that can be verified by rule-based methods, such as text or numbers. Note that it cannot have a decimal point. You can emphasize taking the integer part to avoid this situation;
- Numbers not given in the problem cannot appear in the problem description. All conditions must be given in the problem, and there can be no vague or guessing parts;
- The results of intermediate steps cannot be exposed in the problem. Only the input of nodes with an in-degree of 0 can be given in the problem. The inputs of all other nodes are the outputs of preceding nodes!
- The input of each node in the problem must be used in the sub-problem corresponding to that node. For example, if N5 takes N2 and N4 as input, then the sub-problem corresponding to N5 must use N2 and N4 as conditions and cannot be omitted.
- For nodes with multiple input values, do not simply calculate extremely simple calculations such as N2+N3 or N1 * N4. Instead, flexibly integrate them into atomic tasks. At the same time, if the atomic task itself does not have that many input parameters, you can modify it appropriately, for example:
  - The atomic task is to count the occurrences of a certain character in a string, and there are three inputs N1, N2, N3. You can: convert the number obtained by N1*N2 into a string where 0 corresponds to a and 9 corresponds to j, then splice it with the original string to get a new string. In the new string, count the occurrences of the (N3%10+1)-th letter in the alphabet;
  - The atomic task is factorial calculation, and there are two inputs N3 and N5. You can calculate the factorial of N3%300, take the smaller of the first digit and the last digit of 1000+N5 as start, and the larger as end, and take the (start+1)-th digit to the (end+1)-th digit of the factorial value as the result (the beginning is the 1st digit);
  - Pay special attention: when the output is a number and you want to convert it to a string, you need to design a number -> string conversion method, for example: 0 corresponds to a, 9 corresponds to j; or continuously divide the string by 26, mapping remainders 0-25 to a-z, until the quotient is 0;

Special Attention:

- The problem description must be in English!
- I/O and other sandbox-unsafe operations cannot be used in the verification code!
- The numbers involved in the calculation in the sub-task must be as large as possible to ensure that it is impossible to do without writing code!
- The output code must be runnable, and the answer is obtained in the form of an expression (similar to jupyter) at the end!
- When you call the python tool, you may encounter errors. At this time, you need to adjust the parameters of the problem according to the situation and call the python tool again to get the result!
- In the output json, the code part does not need to be escaped twice, just a normal string!
- In the output json, the category_mapping field should record the category of the atomic task corresponding to each node (such as N0, N1), in the format of a dictionary!
- The output question must be self-contained, all conditions must be complete, including strings, large numbers, etc., allowing no omissions or missing parts, so that people can answer based on the question and get the answer!
- For all cases involving the "nth" position or item, the index starts from 1. This must be clearly written in the final generated question according to the requirements of the atomic task!
- For all problems involving unix timestamps, the timestamp is required to be UTC time. This must be clearly written in the final generated question according to the requirements of the atomic task!
- The node with an in-degree of 0 (source node) also needs to correspond to an atomic task to be solved, that is, the answer to the atomic task is the output of the source node, and the source node cannot be just an input value!!!
- When encountering factorial results, try not to take the last few digits, because basically the last few digits may all be 0. It is better to take the position closer to the front!!!

Atomic tasks are as follows:
[ATOMIC_TASKS_LIST]
The Directed Acyclic Graph combining atomic tasks is as follows:
[MERMAID_DAG]

Figure 4: Prompt for Complex Question Generation (Part 2)

Prompt for DAG Reconstruction

Convert the directed acyclic graph corresponding to the problem-solving process of the following question into mermaid code, requiring:

- Start with "graph TD" on the first line;
- Use expressions like N0, N1 to represent nodes;
- Each subsequent line can only write one edge;
- Use the form "Nx –> Ny" to express an edge;
- Please output in the following format:

```
```mermaid
Your mermaid code
```
```

Question: [QUESTION_TEXT]

Figure 5: Prompt for DAG Reconstruction

Mermaid Example

```
graph TD;
    N0 --> N1;
    N0 --> N3;
    N1 --> N2;
    N1 --> N3;
    N2 --> N3;
    N2 --> N4;
    N3 --> N4;
```

Figure 6: Example of Mermaid DAG Representation

Prompt for Solvability Verification (With Code)

Write Python code to calculate the value of [VARIABLE_NAME] based on the following sub-task.
Sub-task description: [SUBTASK_DESCRIPTION]
Known dependent variable values: [DEPENDENCIES]
Please write complete Python code and output the value of [VARIABLE_NAME] in the form of an expression on the last line (similar to Jupyter). The code should be directly runnable. Note that the code execution has a time limit of 20 seconds, please implement efficient code.
Please output in the following format:

```
```python
Your Python code
```
```

Figure 7: Prompt for Solvability Verification (With Code)

Prompt for Tool Necessity Verification (Without Code)

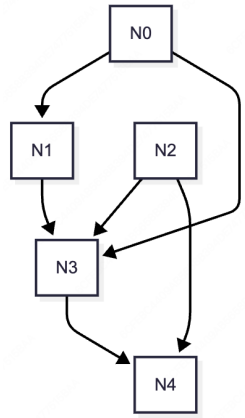
Directly calculate the value of [VARIABLE_NAME] based on the following sub-task.
Sub-task description: [SUBTASK_DESCRIPTION]
Known dependent variable values: [DEPENDENCIES]
Please directly calculate the value of [VARIABLE_NAME], do not write code. Give the answer in the format of \boxed{answer} at the end.

Figure 8: Prompt for Tool Necessity Verification (Without Code)

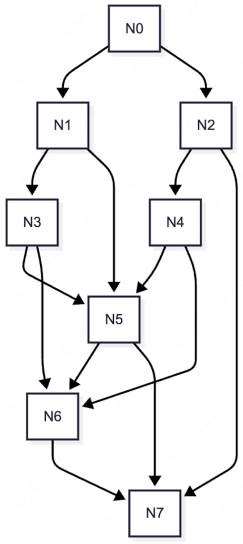
Prompt for Final Question Assembly

[TITLE]
[BACKGROUND_STORY]
Your ultimate task is to find "[FINAL_VARIABLE_NAME]" step by step based on the current clues.
[TASK_DESCRIPTIONS]
Please use the clues above to find the value of "[FINAL_VARIABLE_NAME]" and output it in \boxed{}. At the same time, output the values corresponding to [SUB_TASK_NAMES] as keys in a JSON format. If a value is not obtained, output 'null' for the corresponding value.

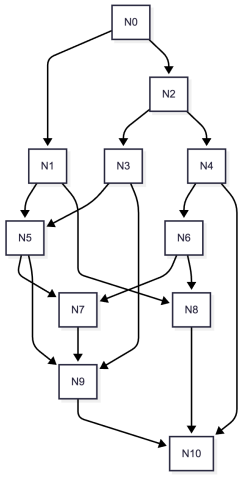
Figure 9: Prompt for Final Question Assembly



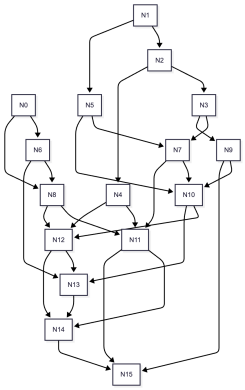
(a) Easy Complexity



(b) Medium Complexity



(c) Hard Complexity



(d) Extreme Complexity

Figure 10: Visualization of Directed Acyclic Graphs (DAGs) demonstrating increasing structural complexity, from simple topologies to extreme density.

C.2 Node-level Correctness

To enable fine-grained analysis, we evaluate the correctness of each intermediate node in the reasoning graph.

Definition. For a given instance x with a ground-truth DAG $G = (V, E)$, each node $v \in V$ represents a sub-problem with a ground-truth value $a_v(x)$. The model is required to output its computed value $\hat{a}_v(x)$ for each node in a structured JSON format. A node v is considered correct if and only if the model's output $\hat{a}_v(x)$ exactly matches the ground-truth value $a_v(x)$ after normalization (e.g., stripping whitespace, unifying numerical formats).

Evaluation Protocol. We employ a strict exact-match criterion for node-level correctness.

- **Parsing**: We parse the model's final JSON output to extract key-value pairs corresponding to each node ID (e.g., "N1", "N2").
- **Matching**: We compare the extracted value with the ground truth. Missing keys, null values, or unparsable formats are treated as incorrect.
- **Dependency Awareness**: While node correctness is evaluated individually, our error attribution analysis (Appendix C.4) considers the topological order. An error in a node is only attributed as a primary failure if all its predecessor nodes were solved correctly (or if it's a leaf node in the dependency context), allowing us to distinguish between root causes and cascading errors.

C.3 Error Attribution Taxonomy

We define the error types as follows:

- **Transcription Error**: The model correctly calculated the value (the correct answer appears in the code execution output) but failed to transcribe it correctly into the final JSON output.
- **Calculation Error**: The model generated code that was executed, but the logic was incorrect, resulting in a wrong value (the wrong value appears in the code execution output).
- **Decryption Logic Error**: In an encrypted sub-task, the model correctly identified and copied the ciphertext into the code, but the decryption logic was flawed, leading to failure.
- **Ciphertext Copy Error**: In an encrypted sub-task, the model failed to correctly copy the long ciphertext string into the code (e.g., typos, truncation), causing the correct decryption algorithm to fail.
- **Refusal (Null)**: The model explicitly output "null" for a node, indicating an admitted inability to solve the sub-problem.
- **No Tool / Hallucination**: The model produced an incorrect value without invoking any tools, or the value did not appear in any tool outputs, suggesting a hallucination or a failure to use the necessary tools.

C.4 Attribution Rules

The attribution rules are applied in the following order to the *first* incorrect node in the topological sort: 1. If the node value is "null" or missing, assign **Refusal (Null)**. 2. If the correct ground-truth value appears in the tool execution outputs for this step, but the JSON output is different, assign **Transcription Error**. 3. If the model's incorrect JSON output value appears in the tool execution outputs, assign **Calculation Error**. 4. If the node involves encryption: a. Check if the ciphertext in the code arguments matches the ground-truth ciphertext. If it matches but the result is wrong, assign **Decryption Logic Error**. b. If the ciphertext in the code is similar (e.g., >50% prefix match, similar length) but not identical to the ground truth, assign **Ciphertext Copy Error**. 5. If none of the above apply (e.g., no tool called, or answer appears nowhere in traces), assign **No Tool / Hallucination**.

C.5 Unanalyzable Cases

We exclude instances from fine-grained error analysis if:

- The model output format is severely broken and the JSON cannot be parsed.
- The DAG structure cannot be reconstructed from the model's output (missing topology).
- The ground truth or model answer is too short (e.g., < 2 characters) to reliably perform string matching for attribution.