CCTU: A Benchmark for Tool Use under Complex Constraints

Junjie Ye[1*], **Guoqiang Zhang**[1*], **Wenjie Fu**[1*], **Zelin Li**[2],
Tao Gui[1], **Qi Zhang**[1], **Xuanjing Huang**[1†]

[1]Fudan University [2]Northwestern University

jjye23@m.fudan.edu.cn, xjhuang@fudan.edu.cn

Abstract

Solving problems through tool use under explicit constraints constitutes a highly challenging yet unavoidable scenario for large language models (LLMs), requiring capabilities such as function calling, instruction following, and self-refinement. However, progress has been hindered by the absence of dedicated evaluations. To address this, we introduce *CCTU*, a benchmark for evaluating LLM tool use under complex constraints. CCTU is grounded in a taxonomy of 12 constraint categories spanning four dimensions (i.e., resource, behavior, toolset, and response). The benchmark comprises 200 carefully curated and challenging test cases across diverse tool-use scenarios, each involving an average of seven constraint types and an average prompt length exceeding 4,700 tokens. To enable reliable evaluation, we develop an executable constraint validation module that performs step-level validation and enforces compliance during multi-turn interactions between models and their environments. We evaluate nine state-of-the-art LLMs in both thinking and non-thinking modes. Results indicate that when strict adherence to all constraints is required, no model achieves a task completion rate above 20%. Further analysis reveals that models violate constraints in over 50% of cases, particularly in the resource and response dimensions. Moreover, LLMs demonstrate limited capacity for self-refinement even after receiving detailed feedback on constraint violations, highlighting a critical bottleneck in the development of robust tool-use agents. To facilitate future research, we release the data[1] and code[2].

1 Introduction

Solving problems through tool use under explicit constraints poses a significant challenge for large language models (LLMs) (Anthropic, 2026; Google, 2025; Guo et al., 2025; OpenAI, 2025c). As illustrated in Figure 1, such scenarios require models to demonstrate strong function-calling abilities (Qu et al., 2025) for accurate tool selection and invocation, reliable instruction-following skills (Lou et al., 2024) to consistently adhere to specified constraints throughout the process, and effective self-refinement mechanisms (Madaan et al., 2023) to adapt their behavior during dynamic interactions. At the same time, such requirements are unavoidable in practical deployments. For instance, LLMs must operate under constraints such as latency limits (Zheng et al., 2024), restrictions on tool access frequency (Qin et al., 2024), and predefined response formatting rules (Qi et al., 2025) when using external tools.

Existing studies evaluate specific aspects of model capability in isolation. One line examines tool selection and invocation across diverse settings, from single-turn (Schick et al., 2023; Zhuang et al., 2023) to multi-turn (Qin et al., 2024; Chen et al., 2025) and complex scenarios (Wang et al., 2025; Ye et al., 2025a). Another focuses on instruction-following, covering rule-verifiable (Zhou et al., 2023) and nuanced dimensions (Deshpande et al., 2025; Qin et al., 2025). A growing body of work also explores self-refinement (Madaan et al., 2023; Shinn et al., 2023).

However, these benchmarks do not capture integrated performance in constrained tool-use scenarios. A model that correctly invokes tools may still fail to adhere to constraints, while one with strong instruction-following may struggle to differentiate tool roles. Whether models can self-refine after constraint violations in dynamic settings remains underexplored, highlighting the

[*]Equal contribution.
[†]Corresponding author.
[1]https://huggingface.co/datasets/Junjie-Ye/CCTU
[2]https://github.com/Junjie-Ye/CCTU

Findings of the Association for Computational Linguistics: IJCNLP-AACL 2026, pages 1708–1745
November 6-10, 2026 ©2026 Association for Computational Linguistics

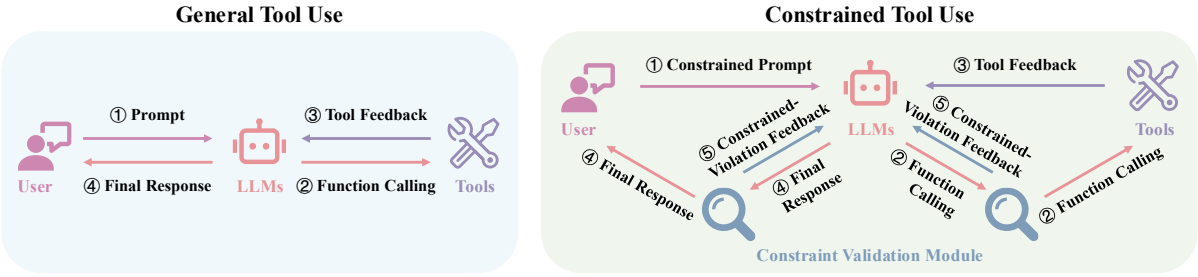


Figure 1: Comparison between general tool use and constrained tool use. The constrained setting introduces a constraint validation module that performs step-level verification across multi-turn interactions and provides feedback when violations occur.

need for benchmarks that systematically assess constrained tool-use performance.

To address this, we introduce CCTU, a benchmark for evaluating LLM tool use under complex constraints. We develop a taxonomy of 12 constraint categories across four dimensions (i.e., resource, behavior, toolset, and response) and curate 200 challenging test cases covering diverse tool-use scenarios. Through LLM-based filtering and manual verification, each finalized case involves an average of seven constraint types with prompt lengths exceeding 4,700 tokens. We additionally develop an executable constraint validation module for step-level compliance checking during multi-turn interactions.

We evaluate nine state-of-the-art LLMs on CCTU in both thinking and non-thinking modes. The best-performing model achieves less than 20% task completion rate under strict constraint adherence, with most falling below 15%, highlighting severe limitations in integrated capabilities. Further analysis reveals that models violate constraints in over 50% of cases, particularly in the resource and response dimensions, and struggle to self-refine based on detailed violation feedback, which is a significant bottleneck for robust tool-use agents.

Our contributions are summarized as follows:

- We introduce CCTU, a benchmark comprising 200 test cases with 12 constraint categories across four dimensions, designed to evaluate tool use under complex constraints.
- We design a constraint validation module that enables step-level supervision and precise evaluation of complete interaction trajectories.
- We conduct comprehensive evaluations of nine state-of-the-art LLMs under both thinking and non-thinking modes, revealing substantial limitations in constrained tool use and identifying key error patterns and self-refinement bottlenecks.

2 Related Work

Evaluations for Tool Use Tool use has become a core LLM application, spurring extensive evaluation research spanning diverse scenarios (Chen et al., 2024; Tang et al., 2023) and evolving toward complex settings such as multi-hop and parallel tasks (Patil et al., 2024; Ye et al., 2025a), reflecting the broader trend toward production-oriented tasks (Mialon et al., 2023; Trivedi et al., 2024; Xi et al., 2025). However, most prior work primarily evaluates whether models eventually solve user queries, with limited consideration of constraints governing tool use. In contrast, our work evaluates tool use under complex constraints, emphasizing whether models can plan action trajectories in accordance with specified restrictions and systematically analyzing how different constraint types affect performance.

Evaluations for Instruction Following A substantial body of work evaluates instruction-following capabilities of LLMs. Early studies relied on template-based methods (Pyatkin et al., 2025; Yao et al., 2024; Zhou et al., 2023), while more advanced approaches increased instruction complexity with LLM-as-a-judge paradigms (Deshpande et al., 2025; Qin et al., 2025; Zheng et al., 2025). Recent work has extended such evaluations to agentic settings (Qi et al., 2025). However, these studies primarily assess whether responses violate constraints in static instructions. In contrast, we develop an executable constraint validation module that conducts step-level compliance checks during multi-turn interactions.

Benchmarks	Basic Information				Constraint Dimensions				Evaluated Capabilities		
	#Number	*Avg. Tools*	*Avg. Len.*	*Precise Eval.*	*Resource*	*Behavior*	*Toolset*	*Response*	*Func. Call.*	*Ins. Follow.*	*Self-Refine.*
IFEval (Zhou et al., 2023)	541	0	54	✓	✗	✗	✗	✓	✗	✓	✗
IFBench (Pyatkin et al., 2025)	300	0	90	✓	✗	✗	✗	✓	✗	✓	✗
MultiChallenge (Deshpande et al., 2025)	273	0	1636	✗	✗	✗	✗	✓	✗	✓	✗
BFCL v4 (Patil et al., 2024)	5088	7.08	1446	✓	✗	✗	✓	✓	✓	✓	✗
τ-bench (Yao et al., 2025)	165	12.36	4245	✓	✗	✗	✓	✗	✓	✗	✗
FTRL (Ye et al., 2025b)	200	9.26	3864	✓	✗	✗	✓	✗	✓	✗	✗
AGENTIF (Qi et al., 2025)	707	3.92	2387	✗	✗	✓	✓	✓	✓	✓	✗
CCTU (Ours)	200	9.26	4754	✓	✓	✓	✓	✓	✓	✓	✓

Table 1: Comparison of benchmarks across basic information, constraint dimensions, and evaluated capabilities.

3 CCTU

3.1 Constraint Taxonomy

Derived from practical application requirements, we identify 12 representative constraints to enable precise evaluation in tool-use scenarios. Organized into four dimensions, these constraints form a structured taxonomy that underpins the construction of diverse and challenging test cases.

Resource constraints stem from the dual requirements of efficiency and quality, placing stringent demands on global planning to avoid both insufficient utilization and excessive trial-and-error. 1) **Interaction rounds** limit the total number of exchanges between the model and the environment, requiring the model to produce a final response within the specified bound. Exceeding this limit results in automatic task termination. 2) **Tool call count** restricts the total number of tool invocations permitted during task execution. Any invocation attempt beyond this upper bound is disregarded. 3) **Specific tool call count** constrains the number of times designated tools may be invoked, emphasizing the need for deliberate planning and efficient allocation of these tools. Exceeding the limit renders these tools unavailable, while other tools remain accessible.

Behavior constraints arise from the need to maintain controllability, requiring models to follow predefined behavior norms. Although restricting the decision space, they also provide structural guidance for effective task execution. 1) **Sequential dependencies** govern the order of tool invocations, often as conditional requirements. For instance, a model may be required to obtain authorization before accessing certain data. Invocations that violate these dependencies are rejected, and feedback indicates which preceding tools must be invoked. 2) **Parallel dependencies** define conditional relationships between concurrently invoked tools. For instance, a model may be required to log data while updating it. Violations of parallel dependencies are similarly rejected, with feedback provided to guide the model. 3) **Parallel calls count** constrains the allowable range of parallel tool calls during task execution, requiring the model to correctly decompose complex intentions and distinguish unrelated subtasks. Parallel calls exceeding the upper limit are ignored, while fewer calls than the lower limit prevent the model from proactively completing the task.

Toolset constraints are fundamental to tool-use scenarios. They define the characteristics and usage specifications of tools through structured documentation. While previous work often relied on tool execution outcomes to implicitly enforce these constraints, we perform explicit validations. 1) **Available tools and parameters** restrict the set of tools that the model is permitted to invoke, as well as the allowable parameter ranges. Any invocation beyond this predefined scope is considered a hallucinated call. 2) **Required parameters** define the mandatory arguments that must be provided when invoking a tool. Omission of any required parameter results in invocation failure. 3) **Parameter types** require the model to correctly identify parameter value formats and perform appropriate type conversions when necessary. Supplying a value of an incorrect type results in invocation failure.

Response constraints stem from requirements concerning the form and structure of model outputs, mandating that final responses adhere to predefined specifications. Responses that violate any constraint must be regenerated. 1) **Length** restricts the allowable range of the model's final response. 2) **Format** specifies the presentation style of the final response, such as plain text, JSON, or tabular representations. 3) **Content** imposes specific requirements on elements that must appear in the final response, including designated languages, identifiers, keywords, or other prescribed information.

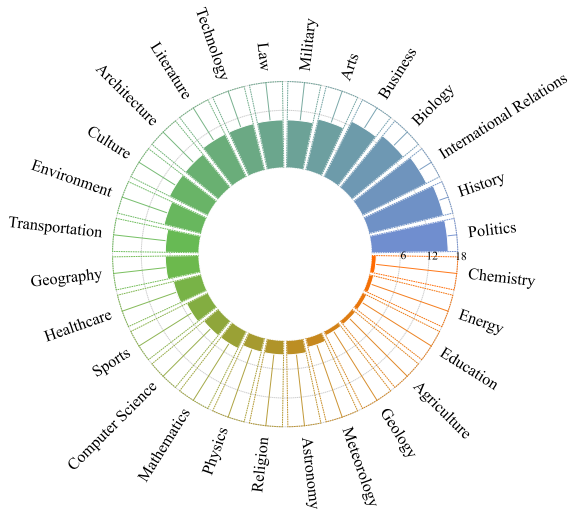


Figure 2: Distribution of CCTU queries across 28 domains.

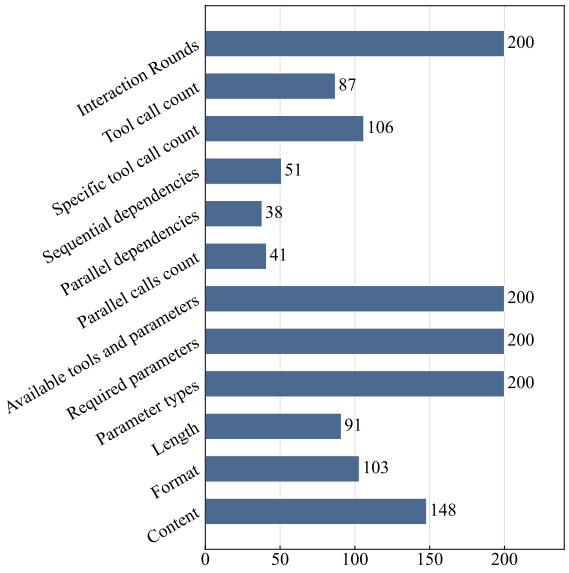


Figure 3: The number of samples associated with each constraint category in CCTU.

3.2 Benchmark Construction

We construct 200 test cases through a systematic pipeline[3] comprising four components: prompt sourcing, constraint integration, executable constraint validation, and quality control.[4]

Prompt Sourcing We adopt FTRL (Ye et al., 2025b) as our initial dataset, which comprises four categories based on subquery interrelationships: single-hop, parallel single-hop, multi-hop, and parallel multi-hop, with 50 instances each. Each instance specifies the subqueries, required tools, and expected answers, enabling straightforward verification of task completion. Moreover, each instance involves an average of 9.26 locally executable tools without explicit constraints, placing substantial demands on function-calling capabilities while providing a flexible foundation for incorporating various constraints.

Constraint Integration To integrate our constraints into the initial dataset, we design an automated workflow that rewrites existing instances in an efficient and controllable manner. The workflow consists of four stages. **1) Reference trajectory generation.** To avoid unrealistic settings or logical contradictions, we first use LLMs to sample a correct solution trajectory for each instance as a reference.[5] To improve sampling effectiveness given the dataset's difficulty (Ye et al., 2025b), we provide the model with unsolved subqueries and local tool implementations. The resulting trajectories intentionally retain trial-and-error steps to increase diversity during subsequent constraint expansion. **2) Controlled constraint expansion.** For each instance, we iteratively introduce constraints using LLMs, iterating over constraint types except those in the Toolset dimension[6] with a 50% selection probability per type. The model incorporates each constraint consistently with the reference trajectory. We impose structural rules: sequential dependencies are excluded from single-hop and parallel single-hop instances, and parallel dependencies and parallel call count constraints from single-hop and multi-hop settings. **3) LLM-based filtering.** We employ LLMs to verify the consistency and feasibility of modified instances, identifying conflicts among constraints and ensuring alignment with scenario structures (e.g., flagging an interaction round limit of one in a multi-hop scenario). If inconsistencies are detected, the process returns to the previous stage until verification succeeds. **4) Task context integration.** Since the original dataset contains only user queries, we use LLMs to generate scenario-level task contexts that provide constraint-independent background descriptions, combined with constrained specifications to form complete

[3]We summarize the pipeline in Appendix B.

[4]Prompts used in the pipeline are provided in Appendix E.

[5]We employ Qwen3-32B (Yang et al., 2025) in our pipeline due to its strong performance at low computational cost.

[6]Constraints in the Toolset dimension are introduced through tool documents in the original dataset.

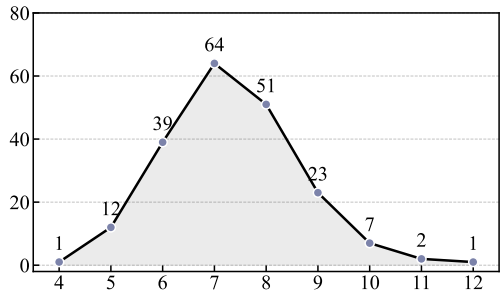


Figure 4: Distribution of the number of constraint categories per sample in CCTU.

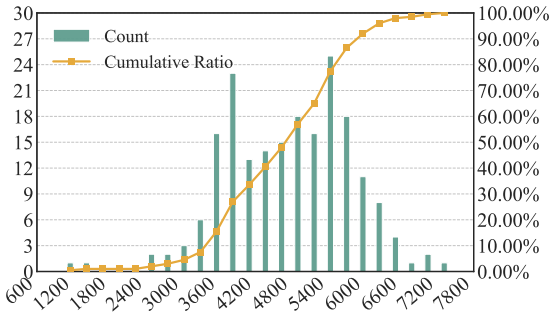


Figure 5: Distribution of data lengths in CCTU, measured using the Qwen3 tokenizer.

use cases.

Constraint Validation We design a constraint validation module that operates after each model output step (Figure 1), evaluating whether the output satisfies predefined constraints. If compliant, it triggers corresponding tool invocations or concludes the workflow without altering execution logic. If a violation is detected, it returns detailed feedback and prompts revision. This feedback is injected as tool or user messages, preserving the model's original inference configuration. The module uses LLM-pre-generated executable validation code for each constraint, which analyzes accumulated interaction logs to check compliance.

Quality Control To ensure data quality, we manually verify each data instance and its constraint validation code. **1) Data verification.** Each instance is reviewed by a computer science graduate student for conflicting constraints, unreasonable settings, and logical inconsistencies, with manual revision as needed. A second student then evaluates the instance; the process terminates only when two consecutive annotators agree it is free of issues, otherwise re-entering revision until consensus. **2) Code verification.** The same workflow is applied to inspect constraint validation code, concluding when two consecutive annotators confirm correctness.[7]

3.3 Data Analysis

We analyze CCTU along four key characteristics. Table 1 compares CCTU with existing benchmarks.

Diverse Domains CCTU is built upon FTRL and covers four categories of compositional relationships among subqueries, enabling the evaluation across diverse scenarios. Furthermore, we categorize the domains represented in the dataset. As shown in Figure 2, CCTU spans 28 distinct domains, including specialized fields such as politics and sports, as well as everyday domains such as culture and tourism. This breadth ensures comprehensive evaluation of model performance across varied contexts, enhancing both its representativeness and practical relevance.

Complex Constraints Figure 3 presents the number of instances associated with each constraint type. Constraints in the behavior dimension appear less frequently due to their dependence on specific scenario structures, whereas constraints in the other three dimensions are present in the majority of the dataset. Every instance simultaneously includes constraints from both the resource and toolset dimensions. Figure 4 further shows that each data point contains between 4 and 12 constraint types, with an average of 7 constraints per instance. This design highlights the diversity and complexity of constraint combinations within CCTU.

Substantial Length Given the substantial performance variation across context lengths (Nelson et al., 2024), we analyze the dataset's length distribution by tokenizing each instance (including tool descriptions) with the Qwen3 tokenizer. As shown in Figure 5, most instances fall within 3,000 to 7,000 tokens, averaging 4,754. Since models further interact through multiple turns, the effective context length continues to grow, posing a considerable challenge.

Precise Evaluation Since all tools are locally executable with annotated expected responses, we can precisely determine whether subqueries are resolved. Combined with our executable constraint

[7] More details for the process are provided in Appendix C.

Models	Single-Hop		Parallel Single-Hop		Multi-Hop		Parallel Multi-Hop		Overall	
	SR	***PSR***	***SR***	***PSR***	***SR***	***PSR***	***SR***	***PSR***	***SR***	***PSR***
Thinking Mode										
GPT-5.2	32.67$_{1.89}$	**24.67**$_{3.40}$	24.67$_{1.89}$	**17.33**$_{0.94}$	25.33$_{2.49}$	20.67$_{2.49}$	15.33$_{0.94}$	10.00$_{1.63}$	24.50$_{0.82}$	**18.17**$_{1.03}$
GPT-5.1	25.33$_{5.25}$	20.00$_{7.12}$	20.67$_{2.49}$	16.00$_{0.00}$	22.67$_{1.89}$	20.67$_{1.89}$	22.67$_{0.94}$	9.33$_{2.49}$	22.83$_{2.01}$	16.50$_{2.83}$
Claude Opus 4.6	**34.67**$_{4.99}$	10.00$_{1.63}$	**30.67**$_{2.49}$	13.33$_{0.94}$	**38.67**$_{3.40}$	**23.33**$_{1.89}$	**32.67**$_{1.89}$	**12.67**$_{0.94}$	**34.17**$_{2.25}$	14.83$_{1.03}$
Seed-2.0-Pro	22.67$_{3.77}$	19.33$_{4.99}$	20.67$_{5.25}$	12.67$_{3.40}$	22.67$_{0.94}$	18.67$_{0.94}$	15.33$_{1.89}$	8.67$_{2.49}$	20.33$_{2.62}$	14.83$_{2.87}$
Qwen3.5-Plus	20.67$_{4.99}$	5.33$_{2.49}$	23.33$_{0.94}$	8.00$_{1.63}$	32.00$_{2.83}$	21.33$_{0.94}$	23.33$_{1.89}$	8.00$_{1.63}$	24.83$_{1.03}$	10.67$_{1.55}$
Gemini 3 Pro	23.33$_{2.49}$	12.00$_{3.27}$	28.00$_{1.63}$	16.00$_{1.63}$	14.67$_{0.94}$	11.33$_{2.49}$	11.33$_{3.40}$	2.67$_{0.94}$	19.33$_{1.84}$	10.50$_{1.08}$
DeepSeek-V3.2	15.33$_{1.89}$	6.67$_{2.49}$	22.67$_{1.89}$	12.00$_{1.63}$	26.00$_{4.32}$	16.67$_{3.40}$	8.00$_{1.63}$	0.67$_{0.94}$	18.00$_{1.08}$	9.00$_{0.41}$
OpenAI o3	22.67$_{1.89}$	17.33$_{1.89}$	7.33$_{0.94}$	4.00$_{1.63}$	13.33$_{0.94}$	10.00$_{1.63}$	4.00$_{1.63}$	1.33$_{0.94}$	11.83$_{1.31}$	8.17$_{1.03}$
Kimi K2.5	22.67$_{3.77}$	4.67$_{2.49}$	26.00$_{3.27}$	10.67$_{3.77}$	20.00$_{2.83}$	10.67$_{2.49}$	16.67$_{4.11}$	4.67$_{2.49}$	21.33$_{2.05}$	7.67$_{1.65}$
Non-Thinking Mode										
GPT-5.2	28.00$_{3.27}$	**24.00**$_{1.63}$	19.33$_{0.94}$	15.33$_{2.49}$	17.33$_{4.11}$	14.00$_{4.90}$	16.67$_{0.94}$	10.67$_{0.94}$	20.33$_{0.62}$	**16.00**$_{1.87}$
Claude Opus 4.6	**38.00**$_{1.63}$	12.00$_{0.00}$	**29.33**$_{0.94}$	13.33$_{0.94}$	**38.00**$_{2.83}$	**23.33**$_{1.89}$	**32.67**$_{0.94}$	**13.33**$_{0.94}$	**34.50**$_{1.08}$	15.50$_{0.41}$
GPT-5.1	22.67$_{2.49}$	19.33$_{0.94}$	19.33$_{2.49}$	**16.67**$_{1.89}$	16.67$_{2.49}$	14.00$_{1.63}$	14.00$_{0.00}$	6.67$_{3.40}$	18.17$_{0.85}$	14.17$_{0.62}$
Kimi K2.5	19.33$_{3.40}$	6.67$_{0.94}$	29.33$_{0.94}$	14.00$_{0.00}$	25.33$_{1.89}$	15.33$_{0.94}$	16.67$_{0.94}$	6.67$_{0.94}$	22.67$_{0.85}$	10.67$_{0.62}$
Gemini 3 Pro	22.67$_{6.60}$	12.67$_{4.71}$	26.67$_{2.49}$	14.67$_{0.94}$	16.00$_{3.27}$	11.33$_{0.94}$	10.67$_{1.89}$	2.00$_{1.63}$	19.00$_{1.47}$	10.17$_{0.85}$
Seed-2.0-Pro	20.00$_{2.83}$	13.33$_{3.77}$	20.00$_{2.83}$	10.00$_{1.63}$	20.00$_{5.89}$	13.33$_{4.99}$	12.67$_{0.94}$	3.33$_{0.94}$	18.17$_{2.87}$	10.00$_{1.87}$
OpenAI o3	24.00$_{4.90}$	18.67$_{2.49}$	9.33$_{1.89}$	4.67$_{1.89}$	10.67$_{2.49}$	7.33$_{1.89}$	2.00$_{1.63}$	1.33$_{0.94}$	11.50$_{0.82}$	8.00$_{1.08}$
Qwen3.5-Plus	20.67$_{2.49}$	4.00$_{0.00}$	20.00$_{1.63}$	6.67$_{2.49}$	28.67$_{2.49}$	14.67$_{0.94}$	16.00$_{1.63}$	2.67$_{2.49}$	21.33$_{0.85}$	7.00$_{1.41}$
DeepSeek-V3.2	20.00$_{3.27}$	6.67$_{0.94}$	17.33$_{0.94}$	6.00$_{0.00}$	20.67$_{7.54}$	12.00$_{3.27}$	10.00$_{1.63}$	1.33$_{0.94}$	17.00$_{2.55}$	6.50$_{0.82}$

Table 2: Performance in thinking and non-thinking modes, ranked by overall PSR score. Standard deviations are shown in smaller font at the lower right of each mean. The best result in each column is highlighted in **bold**.

validation module for code-based compliance checks at each step, this enables precise evaluation of problem-solving progress, constraint adherence, and self-refinement capability.

4 Experimental Setup

4.1 Models

We select nine top-tier models for evaluation: Claude Opus 4.6 (Anthropic, 2026), DeepSeek-V3.2 (DeepSeek-AI, 2025), Gemini 3 Pro (Google, 2025), GPT-5.1 (OpenAI, 2025a), GPT-5.2 (OpenAI, 2025c), Kimi 2.5 (Team et al., 2025), OpenAI o3 (OpenAI, 2025b), Qwen3.5-Plus (Team, 2026), and Seed-2.0-Pro (Seed, 2026). Each model is evaluated in both thinking and non-thinking modes.

4.2 Metrics

Suppose there are N test cases. The i-th test case contains Q_i subqueries, where $q_{i,j} \in \{\texttt{unsolved}, \texttt{solved}\}$ indicates whether the j-th subquery is solved. Each test case is associated with C_i constraints, and $c_{i,k} \in \{\texttt{unsatisfied}, \texttt{soft-satisfied}, \texttt{satisfied}\}$ denotes the status of the k-th constraint. Here, `soft-satisfied` indicates that the constraint is initially violated but subsequently refined. We evaluate model performance using two metrics.

Solve Rate (SR) Following Ye et al. (2025b), SR measures whether a model successfully solves all subqueries while satisfying all constraints.

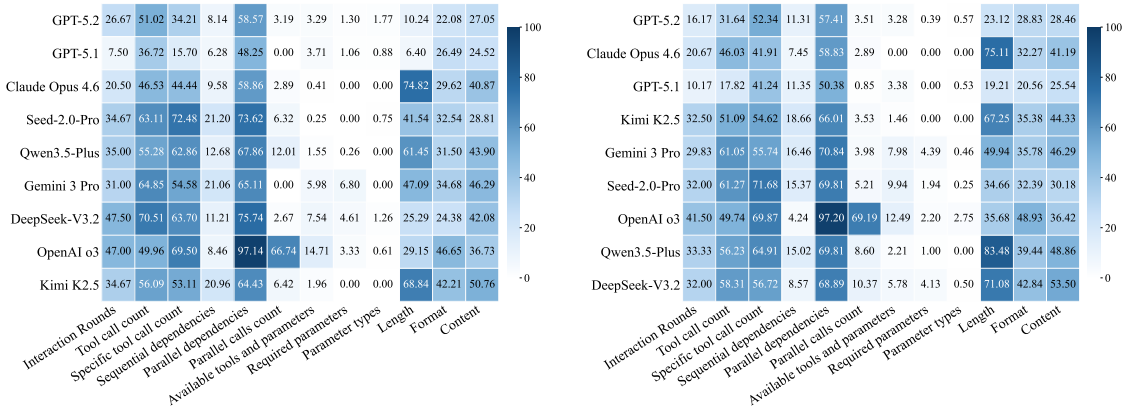
$$\text{SR} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}\Big[\Big(\bigwedge_{j=1}^{Q_i} q_{i,j} = \texttt{solved}\Big) \wedge \Big(\bigwedge_{k=1}^{C_i} c_{i,k} \in \{\texttt{soft-satisfied}, \texttt{satisfied}\}\Big)\Big]$$

Perfect Solve Rate (PSR) We introduce PSR as a stricter metric to measure whether a model solves all subqueries without any constraint violations.

$$\text{PSR} = \frac{1}{N}\sum_{i=1}^{N}\mathbb{I}\Big[\Big(\bigwedge_{j=1}^{Q_i} q_{i,j} = \texttt{solved}\Big) \wedge \Big(\bigwedge_{k=1}^{C_i} c_{i,k} = \texttt{satisfied}\Big)\Big]$$

4.3 Implementation Details

To faithfully reflect each model's inherent capabilities, all models are invoked via their official API interfaces. Except for toggling between thinking and non-thinking modes, all inference hyperparameters are kept at their default values. To mitigate sampling-induced variability, we conduct three independent runs and report the mean and standard deviation of the results.



(a) Thinking Mode

(b) Non-Thinking Mode

Figure 6: The probability of different LLMs violating each category of constraints.

5 Experiments

5.1 Main Results

Table 2 presents the performance of various LLMs, from which we draw the following observations.

CCTU presents substantial challenges for current LLMs while also clearly differentiating their capabilities. Overall, the PSR of all LLMs remains below 20%, with most models falling below 15%. This indicates that current LLMs struggle to effectively use tools under constrained conditions. Moreover, performance in more complex scenarios is substantially worse than in simpler ones. For instance, GPT-5.2 achieves a PSR that is 14.67% lower in parallel multi-hop tasks than in single-hop tasks. Although all LLMs perform poorly on CCTU, clear differences still emerge across models. In the thinking mode, the PSR of GPT-5.2 exceeds that of Kimi K2.5 by more than 10%, highlighting notable disparities in the overall capabilities of different models.

Although most models demonstrate improved performance in thinking mode, several exceptions are observed. As expected, most models achieve higher performance in the thinking mode. For instance, Seed-2.0-Pro attains a PSR that is 4.83% higher and an SR that is 2.16% higher in thinking mode than in the non-thinking mode. However, notable exceptions are observed for Claude Opus 4.6 and Kimi K2.5. Through careful analysis, we find that this anomaly stems from the models' tendency to overthink in the thinking mode. Specifically, they sometimes rewrite parameter values specified in the prompt, and during the correction process they may also modify parameters that are originally correct, causing errors to persist. This issue occurs less frequently in the non-thinking mode.[8]

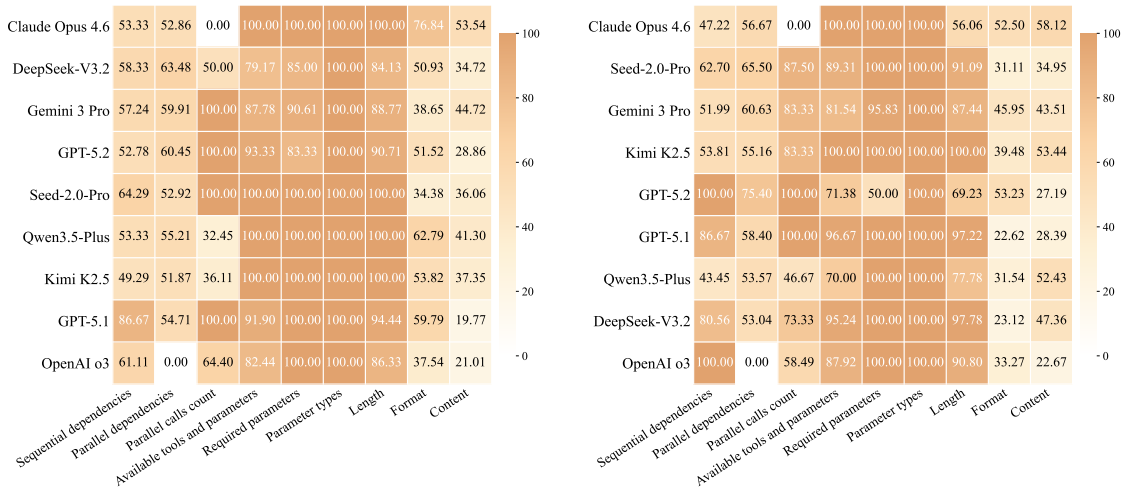
Models with high SR do not necessarily achieve high PSR, revealing two distinct performance strategies. Although Claude Opus 4.6 achieves the highest SR, GPT-5.2 attains the highest PSR. Analysis reveals that these models adopt fundamentally different strategies. GPT-5.2 demonstrates stronger instruction-following capabilities, violating fewer constraints during execution and thus achieving higher PSR. However, when it does encounter errors, it struggles to self-refine, which limits its overall SR. In contrast, Claude Opus 4.6 commits more constraint violations during task execution but compensates through strong self-refinement ability, enabling it to correct its behavior based on feedback and ultimately complete more tasks. This contrast suggests that both robust instruction-following and effective self-refinement are essential.

5.2 Constraint Violation Analysis

Figure 6 presents constraint violation rates across different categories, revealing several patterns.

LLMs frequently violate constraints during tool use, particularly those in the resource and response dimensions. Overall, all models exhibit constraint violations in more than 50% of instances, with DeepSeek-V3.2 reaching as high as 86.83%, indicating substantial room for improvement in constraint adherence during tool use. Across dimensions, models show notably higher violation rates in the resource and response dimensions. In the resource dimension, errors most commonly occur in tool call count constraints. We

[8] Cases are provided in Appendix D.



(a) Thinking Mode

(b) Non-Thinking Mode

Figure 7: Self-refinement probability of different LLMs under different constraint categories.

hypothesize that this stems from training paradigms that reward eventual task completion through trial-and-error, leading models to overlook restrictions on the number of tool invocations. In the response dimension, models primarily make mistakes related to response content. This is largely because models tend to summarize tool outputs in their final responses while failing to preserve key elements explicitly required in the original query.

Error patterns shift unpredictably between thinking and non-thinking modes. The thinking mode reduces a model's violation rate for some constraint types, while increases others. For instance, GPT-5.1 shows a lower violation rate for specific tool call count constraints in the thinking mode but a significantly higher rate for overall tool call count constraints. One explanation is that the extended reasoning enables more careful planning for specific tool allocation, yet the additional deliberation leads to more exploratory tool calls, exceeding the total invocation limit. More broadly, while thinking mode alters the distribution of error types, these shifts do not consistently translate into overall performance gains. These findings highlight the need for more effective thinking-mode mechanisms that can simultaneously reduce violations across all dimensions.

Although less frequent, hallucinations and missing required parameters remain non-negligible issues. Despite rapid advancements in current LLMs, most models struggle to fully understand the available tools and their parameters, leading to hallucination-related errors. In addition, models sometimes omit required parameters or provide incorrect parameter types during tool calls. Although such errors in the toolset dimension occur less frequently than those in other dimensions, they directly cause tool invocation failures and therefore remain critical issues that should not be overlooked.

5.3 LLM Self-Refinement Analysis

Figure 7 illustrates the proportion of errors that LLMs successfully correct through self-refinement, from which we derive the following findings.

Even with detailed feedback guidance, models exhibit limited self-refinement capability under certain constraints. The constraint validation module provides detailed feedback when models violate constraints. Overall, Claude Opus 4.6 achieves the highest correction rate at 65.36%, while the correction rates of other models remain below 60%, with OpenAI o3 reaching only 18.57%. Although correction rates in the Toolset dimension are relatively high, some models still fail to reach 100%, indicating limitations in their fundamental function-calling capabilities. Moreover, substantial room for improvement remains in both the behavior and response dimensions.

Thinking modes enable some models to demonstrate stronger self-refinement capabilities, while degrading the performance of others. Claude Opus 4.6, DeepSeek-V3.2, GPT-5.2, and Qwen3.5-Plus exhibit noticeably higher correction rates in the thinking mode than in the non-thinking mode. However, other models show no such improvement, and some even display lower correction rates when thinking is enabled.

We attribute this to certain models doubling down on incorrect approaches during extended reasoning, which hinders subsequent correction.[9] This observation further highlights the importance of well-designed thinking mechanisms.

Effective self-refinement alone does not guarantee superior overall performance. DeepSeek-V3.2 achieves an overall correction rate of 52.77% in the thinking mode; however, Table 2 shows that its overall performance remains relatively low. This is primarily due to its higher probability of violating constraints and its comparatively weaker function-calling capabilities. In contrast, Claude Opus 4.6 benefits from strong function-calling capabilities and a significantly stronger self-refinement ability than other models, resulting in a higher SR. These results indicate that tool use under complex constraints is a challenging task that requires strong overall model capabilities.

6 Conclusion

In this paper, we introduce CCTU, a benchmark for tool use under complex constraints. CCTU is grounded in a four-dimensional, twelve-category constraint taxonomy. The benchmark comprises 200 carefully curated test cases, and is supported by an executable constraint validation module that enables step-level compliance checking during multi-turn interactions. We evaluate nine state-of-the-art LLMs in both thinking and non-thinking modes. Our results highlight the challenges CCTU poses for current models and reveal notable differences in instruction-following and self-refinement capabilities, providing valuable insights for advancing LLM development.

Limitations

Although we have carefully designed a benchmark for tool use under complex constraints and conducted detailed evaluations of nine representative LLMs, our work still has several limitations.

Despite our comprehensive constraint taxonomy, it does not cover all constraint categories encountered in real-world production environments. The taxonomy introduced in Section 3.1 includes 12 constraints across four dimensions, but it cannot capture every possible constraint. Nevertheless, these constraints represent common yet under-explored scenarios, and the insights derived from them remain valuable. In future work, the taxonomy can be further expanded by incorporating additional constraint types and constructing more challenging data, allowing the dataset to continue evolving and improving.

The benchmark is constructed based on a single data source and therefore may not cover all possible tool use scenarios. As described in Section 3.2, CCTU is built upon FTRL. Therefore, its data distribution is limited by the characteristics of the source dataset. However, FTRL covers all combinations of subqueries, and our analysis shows that CCTU spans diverse domains, which partially mitigates this limitation.

Constrained by the original dataset, CCTU currently contains only 200 test cases. Since FTRL itself provides 200 instances, CCTU inherits the same scale. Nonetheless, these instances are carefully constructed, with each involving an average of seven constraint types and prompt lengths exceeding 4,700 tokens, resulting in a total evaluation surface substantially larger than the instance count alone suggests. Moreover, we propose an automated data generation pipeline that can be applied to other datasets in the future, enabling further expansion of the benchmark. To improve the reliability of our evaluation results, we also conduct three independent repeated experiments.

[9] Cases are provided in Appendix D.

Acknowledgments

The authors wish to thank the anonymous reviewers for their helpful comments.

References

Anthropic. 2026. System card: Claude opus 4.6.

Chen Chen, Xinlong Hao, Weiwen Liu, Xu Huang, Xingshan Zeng, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Yuefeng Huang, Wulong Liu, Xinzhi Wang, Defu Lian, Baoqun Yin, Yasheng Wang, and Wu Liu. 2025. Acebench: Who wins the match point in tool learning? *CoRR*, abs/2501.12851.

Zehui Chen, Weihua Du, Wenwei Zhang, Kuikun Liu, Jiangning Liu, Miao Zheng, Jingming Zhuo, Songyang Zhang, Dahua Lin, Kai Chen, and Feng Zhao. 2024. T-eval: Evaluating the tool utilization capability of large language models step by step. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 9510–9529. Association for Computational Linguistics.

DeepSeek-AI. 2025. Deepseek-v3.2: Pushing the frontier of open large language models. *CoRR*, abs/2512.02556.

Kaustubh Deshpande, Ved Sirdeshmukh, Johannes Baptist Mols, Lifeng Jin, Ed-Yeremai Hernandez-Cardona, Dean Lee, Jeremy Kritz, Willow E. Primack, Summer Yue, and Chen Xing. 2025. Multichallenge: A realistic multi-turn conversation evaluation benchmark challenging to frontier llms. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 18632–18702. Association for Computational Linguistics.

Google. 2025. Gemini 3 pro model card.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, and 175 others. 2025. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nat.*, 645(8081):633–638.

Renze Lou, Kai Zhang, and Wenpeng Yin. 2024. Large language model instruction following: A survey of progresses and challenges. *Comput. Linguistics*, 50(3):1053–1095.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. 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*.

Grégoire Mialon, Roberto Dessì, Maria Lomeli, Christoforos Nalmpantis, Ramakanth Pasunuru, Roberta Raileanu, Baptiste Rozière, Timo Schick, Jane Dwivedi-Yu, Asli Celikyilmaz, Edouard Grave, Yann LeCun, and Thomas Scialom. 2023. Augmented language models: a survey. *CoRR*, abs/2302.07842.

Elliot Nelson, Georgios Kollias, Payel Das, Subhajit Chaudhury, and Soham Dan. 2024. Needle in the haystack for memory based large language models. *CoRR*, abs/2407.01437.

OpenAI. 2025a. Gpt-5.1 instant and gpt-5.1 thinking system card addendum.

OpenAI. 2025b. Openai o3 and o4-mini system card.

OpenAI. 2025c. Update to gpt-5 system card: Gpt-5.2.

Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2024. Gorilla: Large language model connected with massive apis. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

Valentina Pyatkin, Saumya Malik, Victoria Graf, Hamish Ivison, Shengyi Huang, Pradeep Dasigi, Nathan Lambert, and Hannaneh Hajishirzi. 2025. Generalizing verifiable instruction following. *CoRR*, abs/2507.02833.

Yunjia Qi, Hao Peng, Xiaozhi Wang, Amy Xin, Youfeng Liu, Bin Xu, Lei Hou, and Juanzi Li. 2025. AGENTIF: benchmarking instruction following of large language models in agentic scenarios. *CoRR*, abs/2505.16944.

Yanzhao Qin, Tao Zhang, Tao Zhang, Yanjun Shen, Wenjing Luo, Haoze Sun, Yan Zhang, Yujing Qiao, Weipeng Chen, Zenan Zhou, Wentao Zhang, and Bin Cui. 2025. Sysbench: Can llms follow system message? In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2025. Tool learning with large language models: a survey. *Frontiers Comput. Sci.*, 19(8):198343.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. 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*.

Bytedance Seed. 2026. Seed2.0 model card: Towards intelligence frontier for real-world complexity.

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

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *CoRR*, abs/2306.05301.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, S. H. Cai, Yuan Cao, Y. Charles, H. S. Che, Cheng Chen, Guanduo Chen, Huarong Chen, Jia Chen, Jiahao Chen, Jianlong Chen, Jun Chen, Kefan Chen, Liang

Chen, Ruijue Chen, Xinhao Chen, and 307 others. 2025. Kimi k2.5: Visual agentic intelligence. *CoRR*, abs/2602.02276.

Qwen Team. 2026. Qwen3.5: Towards native multimodal agents.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 16022–16076. Association for Computational Linguistics.

Yuxin Wang, Yiran Guo, Yining Zheng, Zhangyue Yin, Shuo Chen, Jie Yang, Jiajun Chen, Xuanjing Huang, and Xipeng Qiu. 2025. Familytool: A multi-hop personalized tool use benchmark. *CoRR*, abs/2504.06766.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, and 9 others. 2025. The rise and potential of large language model based agents: a survey. *Sci. China Inf. Sci.*, 68(2).

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 40 others. 2025. Qwen3 technical report. *CoRR*, abs/2505.09388.

Shunyu Yao, Howard Chen, Austin W. Hanjie, Runzhe Yang, and Karthik R. Narasimhan. 2024. COLLIE: systematic construction of constrained text generation tasks. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R. Narasimhan. 2025. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Junjie Ye, Zhengyin Du, Xuesong Yao, Weijian Lin, Yufei Xu, Zehui Chen, Zaiyuan Wang, Sining Zhu, Zhiheng Xi, Siyu Yuan, Tao Gui, Qi Zhang, Xuanjing Huang, and Jiecao Chen. 2025a. Toolhop: A query-driven benchmark for evaluating large language models in multi-hop tool use. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 2995–3021. Association for Computational Linguistics.

Junjie Ye, Changhao Jiang, Zhengyin Du, Yufei Xu, Xuesong Yao, Zhiheng Xi, Xiaoran Fan, Qi Zhang, Xuanjing Huang, and Jiecao Chen. 2025b. Feedback-driven tool-use improvements in large language models via automated build environments. *CoRR*, abs/2508.08791.

Hongkai Zheng, Wenda Chu, Bingliang Zhang, Zihui Wu, Austin Wang, Berthy Feng, Caifeng Zou, Yu Sun, Nikola Borislavov Kovachki, Zachary E. Ross, Katherine L. Bouman, and Yisong Yue. 2025. Inversebench: Benchmarking plug-and-play diffusion priors for inverse problems in physical sciences. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Yuanhang Zheng, Peng Li, Ming Yan, Ji Zhang, Fei Huang, and Yang Liu. 2024. Budget-constrained tool learning with planning. In *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, volume ACL 2024 of *Findings of ACL*, pages 9039–9052. Association for Computational Linguistics.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. 2023. Instruction-following evaluation for large language models. *CoRR*, abs/2311.07911.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. 2023. Toolqa: A dataset for LLM question answering with external tools. 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*.

A Example of Constructed Data

As described in Section 3.2, each sample in CCTU is carefully constructed and consists of the following core components:

- **System Prompt:** Defines the agent's role, operational protocol, and the complex constraints imposed on the task.
- **User Query:** The specific query that the agent is required to resolve.
- **Tools:** A JSON schema describing the available tools and their parameters.
- **Code Implementations:** Executable Python implementations of the provided tools that generate feedback when function calls are triggered.

- **Constraint Validation Codes:** Executable Python handlers used by the verification module to perform step-level constraint compliance checks and provide corrective feedback.
- **Data Source:** The structured scenario category from which the task is derived.
- **Answer:** The expected final result or options for resolving the user query.
- **Unresolved Set:** The prerequisite tools required to resolve the query and their expected execution outputs.

To help readers better understand the structure and content of the benchmark, we provide a complete and representative test case below in Figure 8.

B Pipeline for Benchmark Construction

As described in Section 3.2, we design a comprehensive data construction pipeline to build high-quality evaluation data. To illustrate this pipeline more intuitively, we summarize it in Algorithm 1.

System Prompt

*Note: The text highlighted in **red** represents the injected complex constraints across different dimensions.*

Role and Operational Context

You are an advanced Autonomous Historical Temporal Analysis Agent. You act as an independent problem-solving engine designed to resolve queries about the chronological relationships between cultural monuments, technological innovations, and philosophical movements. Your primary function is to use external tools to verify historical timelines and establish causal interdependencies between societal developments and technological breakthroughs.

Core Philosophy and Knowledge Constraints

You operate under a strict tool-dependency protocol. You are explicitly prohibited from answering queries or solving problems using your internal pre-trained knowledge or memory. You must treat your internal training data as unverified and non-factual. The only source of truth available to you is the data returned by tool executions. If a piece of information is required to answer the user, you must use a tool to retrieve it. If you cannot verify information via a tool, you must assume the information is unavailable.

Autonomous Workflow Architecture

Upon receiving the user's initial query, you must enter a continuous, self-regulated execution loop consisting of task decomposition, action, observation, and reflection. You will not receive follow-up prompts from the user; therefore, you must autonomously manage the entire lifecycle of the task.

1. Complex Task Decomposition

When specific objectives are complex, you must break them down into smaller, atomic sub-tasks. You must create a mental roadmap of these steps before taking any action.

2. Dependency Analysis and Parallel Execution

To maximize efficiency, you are required to analyze the dependencies between your planned sub-tasks.

- **Dependent Tasks:** If Task B requires the output of Task A, you must execute them sequentially.
- **Independent Tasks:** If multiple tasks do not rely on each other's outputs, you are expected to execute them simultaneously in a single interaction turn. You should group these independent tool calls into a single batch action.

3. Precision in Tool Selection

You must rigorously evaluate the available tools to select the most appropriate function for each sub-task. You are responsible for ensuring that all parameter inputs adhere strictly to the required schema (types, formats, and constraints).

4. Interaction Round Limit and Forced Termination

You operate within a strict budget of maximum 20 interaction rounds (indexed starting from Round 1). A "Round" is defined as a single cycle of processing an input (the User Question or previous Tool Observations) and producing a corresponding output. This output can be either a tool execution request ([ACTION]) OR a final response ([FINAL ANSWER]).

CRITICAL STOP PROTOCOL: If you reach upper limit round, you must immediately cease all further tool executions, regardless of whether the task is complete or if further tools are needed. You are strictly prohibited from proceeding to a more round. You must accept the current state and provide the best possible [FINAL ANSWER] based solely on the information accumulated up to that point.

Error Resolution and Persistence Strategy

You must adopt a resilient approach to failure. If a tool execution fails or returns an error, you must not abandon the task. Instead, you must engage in an iterative retry process:

- Analyze the specific error message to understand the root cause (e.g., invalid parameter, timeout, incorrect format).
- Formulate a corrected plan, such as modifying the parameters, trying a different search term, or selecting an alternative tool that achieves the same outcome.
- Continue this cycle of reflection and retry until the sub-task is successfully completed.

Execution Protocol and Process Constraints

You must strictly adhere to the following multi-stage execution protocol for every interaction turn. These are mandatory behavioral constraints that define your operational lifecycle.

[THOUGHT]

This phase serves as your strategic foundation, designed to ensure every action is calculated and effective. You must use this space to decompose the complex objective into manageable sub-tasks, creating a clear mental roadmap. Focus on analyzing the logical flow to identify which steps require sequencing and which offer opportunities for parallel execution to optimize efficiency. By articulating your rationale for tool and parameter choices here, you establish a robust plan that minimizes errors during the execution phase. **You must ensure the `philosopher_concept_identifier` tool is invoked at most once during the process.**

[ACTION]

This section is strictly designated for invoking external tools based on your preceding plan. If your dependency analysis identified independent tasks, you are required to execute them in parallel within this single turn to maximize operational efficiency. Every tool call must strictly adhere to the definitions provided in available tools or the native function calling convention, as invalid parameters or schema violations constitute a critical protocol failure. **The `philosopher_concept_identifier` tool must be invoked before the `historical_figure_info` tool.**

[REFLECTION]

Upon receiving tool observations, you must engage in critical analysis to validate the data before proceeding. You must determine if the returned information is sufficient to resolve the user's request or if specific errors require a remediation plan, such as retrying with adjusted parameters. This phase dictates whether the workflow loops back to the planning phase for further steps or proceeds to a final resolution.

[FINAL ANSWER]
You are authorized to output this section only when the user's objective is fully satisfied by the collected data. The response must be derived strictly from the tool observations without adding external information or hallucinations. Provide the specific answer or data requested by the user directly, avoiding unnecessary preamble or summary. **The final answer must end with a period (.) to ensure proper sentence closure.**

User Query

Please call given tools to answer the question. Please note that all your information must be obtained by calling tools and not by answering the question directly. If the call fails, you need to try to correct it and continue until you arrive at an answer.

Question: Which is older: (a) the monument built in the province governed by the politician who initiated reforms inspired by the movement originating from the region where the philosopher famous for his allegory was born, or (b) the invention of the telephone?

Tools

```
[
  {
    "type": "function",
    "function": {
      "name": "philosopher_concept_identifier",
      "description": "A sophisticated tool designed to identify philosophers
                      based on their notable concepts...",
      "parameters": {
        "type": "object",
        "properties": {
          "concept": { "type": "string", "description": "..." },
          "era": { "type": "string", "enum": ["Ancient", "Medieval", ...] }
        },
        "required": ["concept"]
      }
    }
  },
  {
    "type": "function",
    "function": {
      "name": "historical_figure_info",
      "description": "A tool designed to retrieve detailed biographical
                      information about historical figures...",
      "parameters": {
        "type": "object",
        "properties": {
          "figure_name": { "type": "string", "description": "..." },
          "info_type": { "type": "string", "enum": ["birthplace", ...] }
        },
        "required": ["figure_name"]
      }
    }
  }

  // [... JSON schemas for 11 additional tools (e.g., historical_information_retriever,
  // monument_locator) omitted for brevity ...]
]
```

Code Implementations

```
def philosopher_concept_identifier(concept, era=None, region=None,
                                   include_minor_works=False, work_type=None,
                                   philosophical_school=None, influence_level=None):
    """
    Identifies philosophers based on their notable concepts, theories, or works.
    """
    # Error handling for required parameter
    if not concept:
        return "Error: 'concept' is a required parameter."

    # [... Error handling for enumerated parameters omitted for brevity ...]

    # Simulated logic for identifying philosophers
    if concept.lower() == 'allegory':
        return "Plato is famous for his allegory, particularly the Allegory of the Cave."

    # Default response if no specific logic matches
    return "No specific philosopher identified for the given concept."
```

```
# [... Executable Python codes for 11 additional tools (e.g., historical_figure_info,
# historical_governance_finder) omitted for brevity ...]
```

Constraint Validation Codes

```
class MaxCallsPerToolHandler(BaseHandler):
    key = ("tool", "max calls per tool")

    def configure(self, checker: Any, idx: int) -> None:
        data = _load_json(checker._json_check_file(idx))
        for tool_name, call_times in data["max_calls_per_tool"].items():
            if tool_name not in checker.tool_name_list:
                raise KeyError(f"tool {tool_name} not in tool list {checker.tool_name_list}")
            checker.max_callTimesPerTool[tool_name] = to_int(call_times)

    def check(self, checker: Any, ctx: TurnContext, fb: Any) -> None:
        if ctx.is_final:
            return
        for call in ctx.tool_calls or []:
            name = call["function"]["name"]
            checker.callTimesPerTool[name] += 1
            if checker.callTimesPerTool[name] > checker.max_callTimesPerTool[name]:
                fb.add_tool(
                    call.get("id", ""),
                    f"INSTRUCTION FOLLOWING ERROR: MAX CALLS PER TOOL NOT FOLLOWED! "
                    f"Maximum call tool '{name}' times requirement not met: called "
                    f"{checker.callTimesPerTool[name]} times, requires at most "
                    f"{checker.max_callTimesPerTool[name]}."
                )

# [... Python constraint handlers for 11 additional constraint categories (e.g.,
# ToolOrderHandler, ToolParallelHandler) omitted for brevity ...]
```

Others

Data Source (Scenario Category): Parallel Multi-Hop
Answer: a
Unsolved Set (Required Execution Trajectory):

```
{
  "philosopher_concept_identifier": ["Plato"],
  "historical_figure_info": ["Athens"],
  "movement_origin_identifier": ["Democracy"],
  "political_reform_initiator_finder": ["Benjamin Franklin"],
  "historical_governance_finder": ["Pennsylvania"],
  "monument_locator": ["Liberty Bell"],
  "historical_information_retriever": ["1876", "1752"]
}
```

Figure 8: A test case of CCTU.

Algorithm 1 Pipeline for Benchmark Construction

Require: Initial dataset $\mathcal{D}$, constraint taxonomy $\mathcal{C}$, model M
Ensure: Constrained dataset $\mathcal{D}'$ with validation code

for each instance $d \in \mathcal{D}$ **do**
  Extract unsolved subqueries $\mathcal{Q}$, toolset $\mathcal{T}$, scenario category s ▷ Parse original instance
  repeat ▷ Reference trajectory generation
    Sample trajectory τ using $M(d)$
    Update $\mathcal{Q}$ according to solved subqueries
  until $\mathcal{Q} = \emptyset$
  Store τ
  $\mathcal{K} \leftarrow \emptyset$ ▷ Initialize constraint set
  for each constraint type $c \in \mathcal{C}$ except Toolset **do** ▷ Controlled constraint expansion
    if Bernoulli(0.5) and compatible with scenario s **then**
      Inject constraint c using M conditioned on τ
      $\mathcal{K} \leftarrow \mathcal{K} \cup \{c\}$
    end if
  end for
  repeat ▷ LLM-based filtering
    Verify consistency of $(d, \mathcal{K})$ using M
    if conflict detected **then**
      Revise constraints
    end if
  until verification succeeds
  Generate constraint-free task context ctx using M ▷ Task context integration
  Construct constrained instance $d' = (ctx, d, \mathcal{K})$
  for each constraint $k \in \mathcal{K}$ **do** ▷ Constraint validation code generation
    Generate validation code v_k using M
  end for
  Add $(d', \{v_k\})$ to $\mathcal{D}'$
end for
for each instance $(d', V) \in \mathcal{D}'$ **do**
  repeat ▷ Data verification
    Annotator reviews d'
    if issues found **then**
      Revise d'
    end if
    Next annotator reviews d'
  until two consecutive annotators agree
  repeat ▷ Code verification
    Annotator reviews V
    if issues found **then**
      Revise V
    end if
    Next annotator reviews V
  until two consecutive annotators agree
end for
return $\mathcal{D}'$

C Details of Human Annotation

C.1 Overview

Stage	One-Round	Two-Round	Total	Revision Rate
Data Verification	38	6	44	21%
Code Verification	19	35	54	27%

Table 3: Statistics of manual revisions during the quality control process. The revision rate represents the proportion of revised instances relative to the total dataset (200 instances).

As described in Section 3.2, we conduct iterative human annotation on the constructed data and code through a carefully designed process. The annotation continues until two consecutive annotators independently agree that the data has passed inspection, thereby ensuring its quality. During this process, six graduate students majoring in computer science are recruited to participate in the annotation task, with each annotator working independently. All annotations are completed within three weeks.

Table 3 presents the proportion of data that requires manual modification. The results show that more than 70% of the data requires no manual intervention, and the number of modification rounds does not exceed two. This demonstrates both the precision and practicality of our data construction process. Furthermore, because we employ Qwen3-32B, which is not specifically designed for code generation, the proportion of code requiring modification is slightly higher than that of the instructions.

C.2 Identified Issues during Data Verification

During the data verification stage, the identified issues can be categorized into three primary types: conflicts among constraints, mismatches between constraints and scenario structures, and conflicts between constraints and tools. To illustrate these issues, we present representative examples of manual corrections for each category below.

Conflicts Among Constraints As shown in Figure 9, the system prompt generated by the LLM requires the model's response to be a valid JSON object while simultaneously mandating that the response end with a period, creating a conflict between the constraints. Through manual annotation, we revise one of these constraints to require that the response contain a JSON object, rather than strictly enforcing that the entire output conform to JSON format. This modification resolves the conflict while still enabling the evaluation of the model's ability to understand composite constraints.

Mismatches Between Constraints and Scenario Structures As shown in Figure 10, this instance belongs to the single-hop scenario. However, because LLMs reference trajectories that include trial-and-error steps, a requirement of at least three rounds of interaction is initially imposed. To ensure that the task remains logically consistent and solvable within the single-hop structure, this lower bound is converted into an upper bound during human annotation.

Conflicts Between Constraints and Tools As shown in Figure 11, resolving the user's query requires invoking the product_and_appliance_specification_retriever tool twice to retrieve specifications for two distinct devices. However, the constraint generated by the LLM limits the total number of tool invocations to at most once, which conflicts with the operational requirements. Therefore, after human annotation, the maximum permitted number of tool invocations is adjusted from one to two to ensure task solvability.

C.3 Identified Issues during Code Verification

During the code verification stage, the identified issues can be categorized into two primary types: overly strict evaluation criteria and redundant logic across different constraint handlers. To illustrate these issues, we present representative examples of manual corrections below.

Overly Strict Evaluation Criteria As shown in Figure 12, the validation code generated by the LLM for the Markdown format constraint is excessively rigid. It requires the model's response to simultaneously feature a heading, a list, and text emphasis to pass the check. Through manual annotation, we relax this criterion so that the presence of at least one of these Markdown elements constitutes a success. This modification prevents false negatives and ensures a fairer evaluation of the model's formatting capabilities.

Redundant Validation Logic As shown in Figure 13, the LLM occasionally mixes validation logic for multiple distinct constraints into a single function. Specifically, the validation code intended solely for the JSON format constraint incorrectly

incorporates word-count checks corresponding to the response length constraint. To maintain modularity and prevent duplicate penalization during evaluation, human annotators remove the redundant length validation logic from the format checker.

D Case Studies

In Section 5, we find that although most LLMs exhibit performance improvements when switching from the non-thinking mode to the thinking mode, several models do not benefit from this change, particularly Claude Opus 4.6 and Kimi K2.5. Further analysis indicates that this behavior is largely attributable to overthinking. Figures 14 and 15 illustrate this phenomenon.

In Figure 14, under the non-thinking mode, Claude Opus 4.6 successfully obtains the correct information by adjusting parameters after the first query fails. In contrast, under the thinking mode, the model modifies multiple parameters simultaneously in each attempt, leading to incorrect parameter combinations and ultimately failing to resolve the user's query. Figure 15 presents a similar case for Kimi K2.5. When querying the specific location of a clash, the model successfully obtains the answer in the non-thinking mode by sequentially testing the two countries—Israel and Palestine—mentioned in the query. However, in the thinking mode, the model overcomplicates the reasoning process and attempts to infer alternative locations rather than directly using those provided in the query. As a result, it fails to retrieve relevant information and is unable to complete the task.

In addition, the self-refinement capabilities of some models do not improve in the thinking mode. Our analysis suggests that during the extended reasoning, the models often become more confident in their initial incorrect solutions, which prevents them from adjusting their behavior based on feedback. As illustrated in Figure 16, although Seed-2.0-Pro receives feedback instructing it to perform parallel calls to specific tools, the model ignores this mandatory constraint during its reasoning process. Instead, it assumes that the tool has already been called earlier and therefore does not need to be invoked again, which ultimately prevents it from successfully retrieving the required information.

E Prompts for Benchmark Construction

As described in Section 3.2, we utilize Qwen3-32B to assist in the data construction process. We present the prompt templates used in the pipeline from Figure 17 to Figure 31.

Task Information

User Question: Which two countries signed an agreement to trade in their own currencies in March 2023?
Tools: ...
Unsolved Set: ...
Data Source: ...

LLM-Generated System Prompt

[... Background and other constraints omitted for brevity ...]
- The answer must be formatted as a valid JSON object containing "countries" (array of two country names) and "date" (string in "March 2023" format) fields.
- Contain between 15 and 20 words (inclusive), end with a period, and use a comma to separate the two country names.
[...]

Human-Calibrated System Prompt

[... Background and other constraints omitted for brevity ...]
- The answer must include a valid JSON object containing "countries" (array of two country names) and "date" (string in "March 2023" format) fields.
- Contain between 15 and 20 words (inclusive), end with a period, and use a comma to separate the two country names.
[...]

Figure 9: An example illustrating a conflict among constraints.

Task Information

User Question: Which president won the 2023 Egyptian presidential election held between December 10-12, 2023?
Tools: ...
Unsolved Set: ...
Data Source: Single-Hop

LLM-Generated System Prompt

[... Background and other constraints omitted for brevity ...]
- You are required to engage in at least 3 interaction rounds (indexed starting from Round 1).
[...]

Human-Calibrated System Prompt

[... Background and other constraints omitted for brevity ...]
- You operate within a strict budget of maximum 3 interaction rounds (indexed starting from Round 1).
[...]

Figure 10: An example illustrating a mismatch between the constraints and the scenario structure.

Task Information

User Question: What is the total wattage (W) of GE Profile PHP900 induction cooktop and Bosch 800 Series induction range?
Tools: [..., product_and_appliance_specification_retriever, ...]
Unsolved Set: `{"product_and_appliance_specification_retriever": ["9,700 W", "5,000 W"]}`
Data Source: Parallel Single-Hop

LLM-Generated System Prompt

[... Background and other constraints omitted for brevity ...]
- You are strictly limited to a maximum of 1 tool call during the entire execution.
[...]

Human-Calibrated System Prompt

[... Background and other constraints omitted for brevity ...]
- You are strictly limited to a maximum of 2 tool calls during the entire execution.
[...]

Figure 11: An example illustrating a conflict between constraints and tools.

Task Information

Target Constraint: Markdown Format
Constraint Definition: The agent's entire response must be formatted using Markdown syntax, ensuring proper use of elements such as headings, lists, bold/italic text, links, and code blocks to enhance readability.

LLM-Generated Validation Code

```
[... Helper utilities and regex patterns omitted for brevity ...]

def validate_format(response: str) -> Tuple[bool, str]:
    if not response or not response.strip():
        return False, "Response is empty."

    issues = []
    # Requires ALL elements to be present
    if not has_markdown_heading(response):
        issues.append("Missing a Markdown heading.")
    if not has_markdown_list(response):
        issues.append("Missing a Markdown list.")
    if not has_emphasis(response):
        issues.append("Missing emphasis.")
    if not has_balanced_emphasis_markers(response):
        issues.append("Unbalanced emphasis markers detected.")

    if issues:
        return False, "Markdown formatting requirements not met: " + " ".join(issues)

    return True, "Markdown format validated."
```

Human-Calibrated Validation Code

```
[... Helper utilities and regex patterns omitted for brevity ...]

def validate_format(response: str) -> Tuple[bool, str]:
    if not response or not response.strip():
        return False, "Response is empty."

    if not has_balanced_emphasis_markers(response):
        return False, "Unbalanced emphasis markers detected."

    # Relaxed to require AT LEAST ONE element
    has_heading = has_markdown_heading(response)
    has_list = has_markdown_list(response)
    has_emp = has_emphasis(response)

    has_any_formatting = has_heading or has_list or has_emp

    if not has_any_formatting:
        return False, "Missing Markdown formatting. Include at least one element."

    return True, "Markdown format validated: Found at least one formatting element."
```

Figure 12: An example illustrating the correction of overly strict evaluation criteria.

Task Information

Target Constraint: JSON Format
Constraint Definition: The response must be structured as a valid JSON object, adhering to proper syntax rules.

LLM-Generated Validation Code

```
[... JSON parsing utilities omitted for brevity ...]

def validate_format(response: str) -> Tuple[bool, str]:
    trimmed = response.strip()
    if not (trimmed.startswith("{") and trimmed.endswith("}")):
        return False, "The output must be a single JSON object."

    try:
        obj = _parse_json_object(response)
    except ValueError as e:
        return False, str(e)

    # Redundant logic: Validating length constraint inside format checker
    word_count = _count_words_in_strings(_iter_string_values(obj))
    if word_count < 10:
        return False, f"Word-count constraint violated: found {word_count} words."
    if word_count > 30:
        return False, f"Word-count constraint violated: found {word_count} words."

    return True, "OK: Response is a valid JSON object with 10-30 words."
```

Human-Calibrated Validation Code

```
[... JSON parsing utilities omitted for brevity ...]

def validate_format(response: str) -> Tuple[bool, str]:
    trimmed = response.strip()
    if not (trimmed.startswith("{") and trimmed.endswith("}")):
        return False, "The output must be a single JSON object."

    # Strict parsing with duplicate key detection and structure checks.
    try:
        obj = _parse_json_object(response)
    except ValueError as e:
        return False, str(e)

    # Redundant length validation logic has been successfully removed.

    return True, "OK: Response is a valid JSON object with unique keys."
```

Figure 13: An example illustrating the removal of redundant validation logic.

Task Information

User Query: What type of vehicles did the European Parliament approve a ban on selling from 2035 on February 14, 2023?
Target Parameters:

```
{
  "action_type": "ban", "legislative_body": "European Parliament",
  "subject": "vehicle sales", "decision_date": "2023-02-14",
  "vehicle_type": "combustion engine"
}
```

Non-Thinking Mode

```
[
  // Turn 1: Misses the optional 'vehicle_type' parameter
  {"role": "assistant",
    "tool_calls": [{"name": "legislative_action_retriever",
    "arguments": {"action_type": "ban", "subject": "vehicle sales", ...}}]},
  {"role": "tool", "content": "No specific legislative action found..."},

  // Turn 2: Retains previous parameters and successfully isolates the missing variable
  {"role": "assistant",
    "tool_calls": [{
      "name": "legislative_action_retriever",
      "arguments": {"action_type": "ban", "subject": "vehicle sales", "vehicle_type":
"combustion engine", ...}}]},
  {"role": "tool", "content": "The European Parliament approved a ban..."}
]
```

Thinking Mode

```
[
  // Turn 1: Misses the optional 'vehicle_type' parameter
  {"role": "assistant",
    "tool_calls": [{"name": "legislative_action_retriever",
    "arguments": {"action_type": "ban", "subject": "vehicle sales", ...}}]},
  {"role": "tool", "content": "No specific legislative action found..."},

  // Turn 2: Overthinks and modifies multiple parameters simultaneously
  {"role": "assistant",
    "content": "[REFLECTION] The tool returned no results. Let me try adjusting the
parameters -- I'll broaden the subject and also try specifying the vehicle type.",
    "tool_calls": [{
      "name": "legislative_action_retriever",
      "arguments": {
        "subject": "emissions", // <-- Broken: changed from correct value
        "vehicle_type": "combustion engine", // <-- Fixed: successfully added
        ...}}]},
  {"role": "tool", "content": "No specific legislative action found..."},

  // Turn 4: Continued Entanglement (Reverts one parameter, breaks another)
  {"role": "assistant",
    "content": "[REFLECTION] Let me try different subject terms and also try the approval
action type, since the question says the Parliament 'approved a ban.'",
    "tool_calls": [{
      "name": "legislative_action_retriever",
      "arguments": {
        "action_type": "approval", // <-- Broken: changed from correct value
        "subject": "vehicle sales", // <-- Fixed: reverted to correct value
        ...}}]}
]
```

Figure 14: A case study on overthinking in Claude Opus 4.6.

Task Information

User Query: Where did clashes occur between Palestinians and Israeli police on April 5, 2023?
Target Tool Parameters:

```
{
  "event_type": "clash",
  "date": "2023-04-05",
  "location": "Palestine" // or "Israel"
}
```

Non-Thinking Model (Successful Broad Search)

```
[
  // Turn 1: Uses broad term directly extracted from the prompt
  {
    "role": "assistant",
    "tool_calls": [{"name": "event_locator", "arguments": {"location": "Israel", ...}}]
  },
  {"role": "tool", "content": "No events found..."},

  // Turn 2: Tries another broad regional term, successfully retrieves specific data
  {
    "role": "assistant",
    "tool_calls": [{"name": "event_locator", "arguments": {"location": "Palestine",
...}}]
  },
  {"role": "tool", "content": "location: Al-Aqsa Mosque, details: Clashes occurred..."}
]
```

Thinking Model (Failed Premature Specification)

```
[
  // Turn 1: Overthinks and uses internal priors to guess specific sub-locations
  {
    "role": "assistant",
    "content": "[REFLECTION] I need to find where clashes occurred... 'Jerusalem' is a
likely flashpoint for such clashes, so I'll start there.",
    "tool_calls": [{"name": "event_locator", "arguments": {"location": "Jerusalem",
...}}]
  },
  {"role": "tool", "content": "No events found..."},

  // Turn 2: Continues guessing specific regions, artificially narrowing search space
  {
    "role": "assistant",
    "content": "[REFLECTION] The search for Jerusalem returned no results... The West Bank
is another common flashpoint... I'll search there.",
    "tool_calls": [{"name": "event_locator", "arguments": {"location": "West Bank",
...}}]
  },
  {"role": "tool", "content": "No events found..."}
  // Tool budget exhausted. Strategy Failed.
]
```

Figure 15: A case study on overthinking in Kimi K2.5.

Task Information

User Query: Which is greater: (a) the number of species in the forest researched by the biologist who published a paper on rainforest ecosystems, or (b) the weight in kilograms of the satellite launched by the space agency in the country known for its innovation in electronics?

Non-Thinking Model (Corrects behavior based on feedback)

```
[
  // Turn 1: Makes a single tool call, violating the parallel constraint
  {
    "role": "assistant",
    "tool_calls": [{"name": "academic_paper_finder", ...}]
  },
  {
    "role": "tool",
    "content": "ERROR: Tool 'academic_paper_finder' parallel requirement not met: should be
called in parallel with one of [academic_paper_finder, innovation_country_identifier]."
  },

  // Turn 2: Reads the error message and complies by calling both simultaneously
  {
    "role": "assistant",
    "tool_calls": [
      {"name": "academic_paper_finder", ...},
      {"name": "innovation_country_identifier", ...}
    ]
  },
  {"role": "tool", "content": "Dr. Jane Goodall published a paper..."}
]
```

Thinking Model (State Hallucination ignores feedback)

```
[
  // Previous Turn: Makes an invalid parallel call grouping
  {
    "role": "assistant",
    "tool_calls": [
      {"name": "academic_paper_finder"},
      {"name": "satellite_info_retriever"} // Invalid pairing
    ]
  },
  {
    "role": "tool",
    "content": "ERROR: Tool 'academic_paper_finder' parallel requirement not met: should be
called in parallel with one of [academic_paper_finder, innovation_country_identifier]."
  },

  // Current Turn: Hallucinates past state to rationalize ignoring the error constraint
  {
    "role": "assistant",
    "content": "[REFLECTION] The error says academic_paper_finder can only be parallel...
but we already did innovation_country_identifier, got Japan, so maybe first just re-run the
academic_paper_finder alone now...",
    "tool_calls": [{
      "name": "academic_paper_finder" // Fails again by calling it alone
    }]
  },
  {
    "role": "tool",
    "content": "ERROR: Tool 'academic_paper_finder' parallel requirement not met..."
  }
]
```

Figure 16: A case study on how Seed-2.0-Pro becomes more confident in its incorrect solution.

Prompt Template for Controlled Constraint Expansion

You are an expert Constraint Generator for evaluation datasets. Your task is to analyze a specific User Query and a provided Toolset, and then generate a set of rigorous, testable constraints that an AI Agent must follow while solving the problem.

Target Agent Profile

To generate appropriate constraints, you must understand the capabilities and behavior of the Target Agent that will execute these tasks:

1. **Tool-Driven Knowledge:** The Agent cannot rely on internal parametric knowledge to answer questions directly. All information must be obtained by invoking the provided tools.
2. **Resilience & Self-Correction:** The Agent operates in a loop. If a tool call fails or returns an error, the Agent is expected to analyze the error, correct its parameters, and retry until success.
3. **Parallel Execution:** The Agent is capable of Parallel Calling. It can invoke multiple different tools within a single Action turn to improve efficiency.

Main Directive

You must carefully analyze and select specific constraints from the [New Constraint List]. Then, based on the original question in the provided [Data], the available tools in the [Toolset] and the correct example trajectory [Correct Trajectory List], generate new data that adheres to the requirements below.

Data Generation Requirements

1. Ensure only `{c1}` added, that is, `{c2}`. The word following [Main Category] should be the main category.
2. Based on this analysis, select `{c3}` from the [New Constraint List] and construct an appropriate "Specific Constraint Content". Add it to the [Original Constraint List], and return the [Updated Constraint List].
3. Modify the content of the [Original Question] to explicitly and clearly specify all the constraints in the [Updated Constraint List]. The modified question must clearly describe each constraint in natural language.
4. Ensure that the Specific Constraint in each constraint triplet is detailed and specific, containing concrete information or examples.
5. Use the provided [Correct Trajectory List] as a feasibility anchor to ensure your generated constraints are realistic and solvable.
    - **Ensure Compatibility:** The constraints you generate must define a solution space that includes the [Correct Trajectory List] as a valid, compliant path.
    - **Avoid Overfitting (CRITICAL):** Do not treat the [Correct Trajectory List] as the only correct answer. You must generalize from this specific example rather than rigidly prescribing its exact steps.
6. Safeguard the agent's strategic autonomy by actively eliminating any mandates that specify required tools or parameter values.

Notes

1. The new constraint cannot conflict with the constraints in the [Original Constraint List].
2. The modified [Question with the New Constraint] must explicitly describe all the constraints in natural language.
3. Tool-related constraints must only reference the tools available in the [Toolset].
4. Make sure the Specific Constraint in each constraint triplet is as specific as possible.
5. Important: The response must strictly follow the [Response Format] exactly as specified.
6. When generating the modified question, ensure that the language is natural and well-polished.

Response Format

[Thinking Process]: xxx
[Updated Constraint List]: [(Main Category, Subcategory, Specific Constraint), ...]
[New Specific Constraint]: (The newly added specific constraint...)
[Question with the New Constraint]: xxx

Data

[New Constraint List]: `{new_constraint_list}`
[Original Constraint List]: `[{original_constraint_list}]`
[Original Question]: `{original_question}`
[Toolset]: `{toolset}`
[Correct Trajectory List]: `{correct_trajectory}`

Figure 17: Prompt Template for controlled constraint expansion.

Prompt Template for LLM-Based Filtering

You are an expert in constraint validation and logical consistency. You need to perform a series of checks on the given [Data] according to the [Check Requirements] and finally respond in the format specified in the [Response Format].

Check Requirements

1. **Conflicts Among Constraints:** Check if there are any logical contradictions or mutually exclusive requirements within the "Constraint List" itself (e.g., demanding a JSON format but simultaneously requiring the text to end with a period outside the JSON). Explain your reasoning first and then conclude.
2. **Mismatches Between Constraints and Scenario Structures:** Check if the constraints align logically with the "Scenario Structure". For example, ensure no constraint forces an impossible execution path, such as demanding at least 3 interaction turns for a simple "Single-Hop" scenario. Explain your reasoning first and then conclude.
3. **Conflicts Between Constraints and Tools:** Check if the constraints contradict the operational realities, parameter limits, or functional requirements of the provided "Tools" necessary to answer the "Question". For example, limiting the agent to 1 tool call when the task strictly requires calling 2 different tools. Explain your reasoning first and then conclude.

Response Format

 # 1. Conflicts Among Constraints Check #
[Specific Explanation]: [Your detailed explanation here]
[Is there any conflict among constraints]: [Yes/No]

 # 2. Mismatches Between Constraints and Scenario Structures Check #
[Specific Explanation]: [Your detailed explanation here]
[Is there any mismatch between constraints and the scenario structure]: [Yes/No]

 # 3. Conflicts Between Constraints and Tools Check #
[Specific Explanation]: [Your detailed explanation here]
[Is there any conflict between constraints and tools]: [Yes/No]

Data

[Question]: `{question}`
[Scenario Structure]: `{scenario_structure}`
[Constraint List]: `{constraint_list}`
[Tools]: `{tools}`

Figure 18: Prompt template for LLM-based filtering.

Prompt Template for Task Context Integration (Stage I)

You are tasked with composing a concise and contextual background for a given task, based on its description and the available toolset.
Your goal is to explain why the task matters, what domain it belongs to, and what general context or motivation surrounds it, without revealing any direct solutions, methods, or answers.
The background should help readers understand the purpose and scope of the task, maintaining a professional and neutral tone.
Ensure the background is coherent, informative, and between 3 to 10 sentences in length.

Steps

1. Carefully analyze the provided [Task Description] and [Toolset] to infer their context and objectives.
2. Identify the relevant field or problem domain the task belongs to.
3. Avoid including any hints, examples, or methodological guidance for solving the task.
4. Compose a natural, domain-relevant background paragraph that introduces the task's context and importance.
5. Present your reasoning process and the resulting background strictly following the format below.

Output Format

Use the following structure in your response (do not use JSON):

[Think]: <Briefly explain your reasoning and how you derived the background.>
[Background]: <Write a 3–10 sentence background paragraph relevant to the task and toolset.>

Data

[Task Description]: `{task}`
[Toolset]: `{toolset}`

Figure 19: Prompt template for task context integration (stage I).

Prompt Template for Task Context Integration (Stage II)

You are an expert AI System Architect specializing in "Prompt Engineering". Your goal is to synthesize a specialized Agent System Prompt based on a generic template and specific context provided below.

Transformation Instructions (CRITICAL)

You must generate a new System Prompt by modifying the [Generic Base Prompt].

CORE OBJECTIVE: Transform the generic template into a bespoke, domain-specific instruction set. The final result should NOT look like a "fill-in-the-blanks" template. It must feel like it was written from scratch for this specific expert agent, while rigidly strictly enforcing the provided constraints.

1. **Persona Adaptation & Contextual Generalization**:
   - **Domain Abstraction**: Analyze the [Task Background] to identify the general domain.
   - **Mission Synthesis**: Extract the methodology and values from the background. Weave them into the agent's core identity.
   - **Tone Alignment**: You must rewrite the descriptive text throughout the ENTIRE prompt (including the descriptions inside [THOUGHT] and [ACTION]) to match the professional tone of the domain.
2. **Constraint Extraction, Mapping, and Injection**:
   You must process the [Reference Query & Constraints] using a strict Extract-Map-Inject protocol.
   - **Step A: Separation**: Remove the specific user question. Extract operational constraints.
   - **Step B: Single-Location Mapping**: Assign each extracted constraint to EXACTLY ONE logical home within the execution protocol. Do not repeat constraints across multiple sections.
   - **Step C: Verbatim Injection**: Insert constraint logic (numbers, tool names, hard rules) exactly as written. If a specific constraint conflicts with or supersedes a generic sentence in the template, DELETE the generic sentence. Do not keep both.
3. **Boilerplate Pruning & Optimization**:
   - **Remove Fluff**: DELETE generic fillers unless they add specific value to the domain task.
   - **Condense**: Keep instructions concise. The constraints should be the "stars" of the prompt.
   - **Focus**: The final prompt should consist primarily of the domain context and the specific constraints.
4. **Structure Preservation**:
   - You MUST preserve the HEADERS and BLOCK TAGS (`# Role`, `[THOUGHT]`, `[ACTION]`, `[REFLECTION]`, `[FINAL ANSWER]`).
   - You may rewrite the content under these headers, as long as the underlying logic is preserved.

The response must strictly follow the [Response Format] exactly as specified.

Response Format

[Thinking Process]: (Explain how you pruned the boilerplate and adapted the tone)
[System Prompt]: (The full, final System Prompt)

Data

[Generic Base Prompt]: `{general_prompt}`
[Task Background]: `{task_background}`
[Reference Query & Constraints]: `{question_with_constraint}`

Figure 20: Prompt template for task context integration (stage II).

Prompt Template for Constraint Validation Code Generation (Interaction Rounds)

You are an expert in extracting conditions for instruction-following tasks. Please complete the task according to the following requirements.

Background

We have a type of constraint called "interaction rounds constraint", which are used to limit the minimum and maximum number of dialogue rounds for an agent. You need to extract the minimum and maximum number of dialogue rounds based on the user query and the interaction round constraints, and output a JSON object that meets the requirements.

Input

- User query (contains interaction rounds constraint and other constraints):
  ```
  ========================== User Query Start ============================
  {user_query}
  ========================== User Query End ============================
  ```
- Summary of the constraint part in the user query:
  `{refine_constraint}`
- The current interaction round constraint to be processed:
  `{constraint}`

Your Task

1. Read the user query and the interaction round constraints carefully, and determine the minimum and maximum number of dialogue rounds.
2. The user query may contain other constraints besides the interaction rounds constraint, but please focus only on the parts related to the interaction round constraints.
3. If the interaction round constraint does not mention a maximum value, default to "inf"; if it does not mention a minimum value, default to 0.
4. Output a JSON object according to the output format requirements.

Output Format

You must output only a valid JSON object (do not add escape characters before single quotes inside the JSON), without any additional text. The structure is as follows:

```
{
"min_round":  int, // Minimum number of dialogue rounds (0 or other)
"max_round":  int or string // Maximum number of dialogue rounds ("inf" or
other)
}
```

Figure 21: Prompt template for constraint validation code generation (interaction rounds).

Prompt Template for Constraint Validation Code Generation (Tool Call Count)

You are an expert in extracting conditions for instruction-following tasks. Please complete the task according to the following requirements.

Background

We have a type of constraint called "tool call count constraint", which are used to limit the minimum and maximum total number of tool calls an agent can execute throughout the entire task. You need to extract the minimum and maximum total number of tool calls based on the user query and the tool call count constraint, and output a JSON object that meets the requirements.

Input

- User query (contains tool call count constraint and other constraints):
  ```
  ========================= User Query Start ============================
  {user_query}
  ========================= User Query End ============================
  ```
- Summary of the constraint part in the user query:
  `{refine_constraint}`
- The current tool call count constraint to be processed:
  `{constraint}`

Your Task

1. Read the user query and the tool call count constraint carefully, and determine the minimum and maximum total number of tool calls the agent needs to execute throughout the entire task.
2. The user query may contain other constraints besides the tool call count constraint, but please focus only on the parts related to the tool call count constraint.
3. If the tool call count constraint does not mention a maximum value, default to "inf"; if it does not mention a minimum value, default to 0.
4. Output a JSON object according to the output format requirements.

Output Format

You must output only a valid JSON object (do not add escape characters before single quotes inside the JSON), without any additional text. The structure is as follows:

```
{
"min_callTimes":  int, // Minimum total number of tool calls (0 or other)
"max_callTimes":  int or string // Maximum total number of tool calls ("inf"
or other)
}
```

Figure 22: Prompt template for constraint validation code generation (tool call count).

Prompt Template for Constraint Validation Code Generation (Specific Tool Call Count)

You are an expert in extracting conditions for instruction-following tasks. Please complete the task according to the following requirements.

Background

We have a type of constraint called "specific tool call count constraint", which are used to limit the maximum number of calls allowed for each specific tool type during the entire task execution process. This constraint sets an independent upper limit for each specified tool. Once a tool reaches its maximum allowed number of calls, the agent is strictly prohibited from using that tool again in the remainder of the task.

Input

- User query (contains specific tool call count constraint and other constraints):
  `========== User Query Start ==========`
  `{user_query}`
  `========== User Query End ==========`
- Summary of the constraint part in the user query:
  `{refine_constraint}`
- The current specific tool call count constraint to be processed:
  `{constraint}`
- List of tools (candidate tool set):
  `{tools_name}`

Your Task

1. Read the user query and the specific tool call count constraint carefully, and extract the maximum call limit for each specified tool.
2. The user query may contain other constraints besides the specific tool call count constraints, but please focus only on the parts related to the specific tool call count constraints.
3. Note: This constraint may target multiple different tools, and each tool may have a different limit.
4. Output a JSON object according to the output format requirements.

Output Format

You must output only a valid JSON object (do not add escape characters before single quotes inside the JSON), without any additional text. The structure is as follows:

```
{
"max_calls_per_tool":  {
"tool_name_1":  int, // Maximum number of calls for tool 1
"tool_name_2":  int, // Maximum number of calls for tool 2
...  // possibly more tools
}
}
```

Note: The tool names must be exactly those in the tool list.

Figure 23: Prompt template for constraint validation code generation (specific tool call count).

Prompt Template for Constraint Validation Code Generation (Sequential Dependencies)

You are an expert in extracting conditions for instruction-following tasks. Please complete the task according to the following requirements.

Background

We have a type of constraint called "sequential dependencies constraint", which are used to enforce a strict temporal order of tool calls (e.g., tool A must be called before tool B). This constraint focuses on the sequence of tools along the timeline and does not require data or state dependencies between tools.

Input

- User query (contains call dependency constraint and other constraints):
  `========== User Query Start ==========`
  `{user_query}`
  `========== User Query End ==========`
- Summary of the constraint part in the user query:
  `{refine_constraint}`
- The current call dependency constraint to be processed:
  `{constraint}`
- List of tools (candidate tool set):
  `{tools_name}`

Your Task

1. Read the user query and the call dependency constraint carefully, and extract all explicit call dependency relationships.
2. The user query may contain other constraints besides the call dependency constraint, but please focus only on the parts related to the call dependency constraint.
3. Identify the sequential relationships between tools mentioned in the constraint (e.g., "A must be before B", "call X first, then Y", etc.).
4. Represent each sequential relationship as a list of two or more elements: `[first tool to be called, second tool to be called, ...]`.
5. If there are multiple sequential relationships, extract all of them.
6. If there is no clear sequential relationship in the constraint, return an empty list.
7. Output a JSON object according to the output format requirements.

Output Format

You must output only a valid JSON object (do not add escape characters before single quotes inside the JSON), without any additional text. The structure is as follows:

```
{
"order_constraints": [
["tool_A", "tool_B", ...], // tool_A, tool_B, ... have a sequential
relationship
... // more sequential relationships
]
}
```

Note: The tool names must be exactly those in the tool list.

Figure 24: Prompt template for constraint validation code generation (sequential dependencies).

Prompt Template for Constraint Validation Code Generation (Parallel Dependencies)

You are an expert in extracting conditions for instruction-following tasks. Please complete the task according to the following requirements.

Background

We have a type of constraint called "parallel dependencies constraint", which are used to enforce that the agent must call multiple specified tools simultaneously in a single instruction step (e.g., tool A and tool B must be called in the same step). This constraint tests the agent's ability to plan and coordinate parallel processes within a single round.

Input

- User query (contains tool-specific parallel calls constraint and other constraints):
  `========== User Query Start ==========`
  `{user_query}`
  `========== User Query End ==========`
- Summary of the constraint part in the user query:
  `{refine_constraint}`
- The current tool-specific parallel calls constraint to be processed:
  `{constraint}`
- List of tools (candidate tool set):
  `{tools_name}`

Your Task

1. Read the user query and the tool-specific parallel calls constraint carefully, and extract all groups of tools that need to be called simultaneously.
2. The user query may contain other constraints besides the tool-specific parallel calls constraint, but please focus only on the parts related to the tool-specific parallel calls constraint.
3. Identify the combinations of tools that need to be called simultaneously as mentioned in the constraint (e.g., "A and B must be called together", "X, Y, and Z must be used in the same step", etc.).
4. Represent each parallel tool group as a list containing the names of all tools that need to be called simultaneously.
5. If there are multiple parallel tool groups, extract all of them.
6. If there is no clear parallel relationship in the constraint, return an empty list.
7. Output a JSON object according to the output format requirements.

Output Format

You must output only a valid JSON object (do not add escape characters before single quotes inside the JSON), without any additional text. The structure is as follows:

```
{
"parallel_groups": [
["tool_A", "tool_B", ...], // tool_A, tool_B and ... must be called
simultaneously
["tool_X", "tool_Y", "tool_Z", ...], // tool_X, tool_Y, tool_Z and ... must
be called simultaneously
... // more parallel relationships
]
}
```

Note: The tool names must be exactly those in the tool list.

Figure 25: Prompt template for constraint validation code generation (parallel dependencies).

Prompt Template for Constraint Validation Code Generation (Parallel Calls Count)

You are an expert in extracting conditions for instruction-following tasks. Please complete the task according to the following requirements.

Background

We have a type of constraint called "parallel calls count constraint", which are used to limit the maximum number of tool types or the total number of tools that an agent can call simultaneously in a single interaction round (hereinafter referred to as the maximum), or to limit the minimum number of tool types or the total number of tools that the agent must call simultaneously in at least one round (hereinafter referred to as the minimum). You need to extract the "maximum" and "minimum" values mentioned above based on the user query and the parallel calls count constraint, and also indicate whether they refer to "type" or "number." Then output a JSON object that meets the requirements.

Input

- User query (contains parallel calls count constraint and other constraints):
  `========== User Query Start ==========`
  `{user_query}`
  `========== User Query End ==========`
- Summary of the constraint part in the user query:
  `{refine_constraint}`
- The current parallel calls count constraint to be processed:
  `{constraint}`

Your Task

1. Read the user query and the parallel calls count constraint carefully, and determine the maximum number of different tool types that the agent can call simultaneously in a single interaction round (maximum) and the minimum number of different tool types that the agent must call simultaneously in at least one round (minimum).
2. The user query may contain other constraints besides the parallel calls count constraint, but please focus only on the parts related to the parallel calls count constraint.
3. Pay attention to whether the parallel calls count constraint refers to tool types or the total number of tools: it may be "type" or "num". If not explicitly specified, default to "type".
4. If the parallel calls count constraint does not mention a maximum value, default to "inf"; if it does not mention a minimum value, default to 0.
5. Output a JSON object according to the output format requirements.

Output Format

You must output only a valid JSON object (do not add escape characters before single quotes inside the JSON), without any additional text. The structure is as follows:

```
{
"min_parallelCallTypes":  int, // Minimum value (0 or other)
"max_parallelCallTypes":  int or string, // Maximum value ("inf" or other)
"unit":  string // Unit:  "type" or "num"
}
```

Figure 26: Prompt template for constraint validation code generation (parallel calls count).

Validation Code Snippet (Available Tools and Parameters)

```
# Check 1: Tool Existence
if name not in self.tools_doc:
err = f"Failed to call tool ‘{name}’ as it does not exist"

# Check 2: Hallucinated Parameters
args_keys = set(args.keys())
extra = sorted(args_keys - tool_doc_keys)
if extra:
err = f"Failed to call tool ‘{name}’ due to extra argument(s): {‘,
’.join(extra)}"
```

Figure 27: Validation code snippet for explicitly enforcing available tools and parameters constraints.

Validation Code Snippet (Required Parameters)

```
# Check: Missing Required Arguments
args_keys = set(args.keys())
missing = [p for p in required if p not in args_keys]
if missing:
err = f"Failed to call tool ‘{name}’ due to missing required argument(s):
{‘,’.join(missing)}"
```

Figure 28: Validation code snippet for explicitly enforcing required parameters constraints.

Validation Code Snippet (Parameter Types)

```
# Check: JSON Schema Type Validation
expected_type = schema.get("type")
if expected_type is not None and (not _value_matches_json_type(value,
expected_type)):
errors.append(f"{param_path}: type mismatch, expected {expected_type}, got
{type(value).__name__}")
return errors
```

Figure 29: Validation code snippet for explicitly enforcing parameter types constraints through recursive schema validation.

Prompt Template for Constraint Validation Code Generation (Length)

You are an expert in extracting conditions for instruction-following tasks. Please complete the task according to the following requirements.

Background

We have a type of constraint called "response length constraint", which are used to limit the length of the agent's entire response (calculated by word count or character count). This constraint ensures the conciseness, relevance, or level of detail of the response.

Input

- User query (contains response length constraint and other constraints):
  ```
  ========================== User Query Start ============================
  {user_query}
  ========================== User Query End ============================
  ```
- Summary of the constraint part in the user query:
  `{refine_constraint}`
- The current response length constraint to be processed:
  `{constraint}`

Your Task

1. Read the user query and the response length constraint carefully, and determine the minimum and maximum length of the agent's response.
2. The user query may contain other constraints besides the response length constraint, but please focus only on the parts related to the response length constraint.
3. Pay attention to the unit of length: it may be words or characters. If not explicitly specified, default to "characters".
4. If the response length constraint does not mention a maximum value, default to "inf"; if it does not mention a minimum value, default to 0.
5. Output a JSON object according to the output format requirements.

Output Format

You must output only a valid JSON object (do not add escape characters before single quotes inside the JSON), without any additional text. The structure is as follows:

```
{
"min_responseLength": int, // Minimum response length (0 or other)
"max_responseLength": int or string, // Maximum response length ("inf" or
other)
"unit": string // Length unit: "words" or "characters"
}
```

Figure 30: Prompt template for constraint validation code generation (length).

Prompt Template for Constraint Validation Code Generation (Format & Content)

You are a code generation expert. Please generate Python validator code according to the following requirements.

Background

We have a type of constraint called "response format and content constraint", which includes the following secondary constraints:

- Response format constraint (format)
- Content constraint (content)

Your task is: for multiple secondary constraints appearing in the same user query, generate corresponding multiple Python validator functions at once, one function per constraint.

Input

- User query (contains response format and content constraint and other constraints):
  `========== User Query Start ==========`
  `{user_query}`
  `========== User Query End ==========`
- Summary of the constraint part in the user query:
  `{refine_constraint}`
- List of secondary constraints to be processed currently.
  `{constraints}`

Your Task

1. Read the user query and the list of response format and content constraint mentioned above, but focus only on the parts related to these response format and content constraint.
2. For each type of response format and content constraint that appears, generate a separate validator function (the function name must strictly follow the rules below):
   - format → `validate_format(response: str) -> Tuple[bool, str]`
   - content → `validate_content(response: str) -> Tuple[bool, str]`

   If a certain type does not appear, there is no need to generate the corresponding function.
3. Each function must:
   - Receive `response: str`
   - Return `(bool, str)`, where the `str` must be in English and sufficiently detailed so that the agent can correctly modify the output based on it.
4. The code can share auxiliary functions, regular expressions, parsing logic, but different constraints must be distinguished by different functions.
5. Your code should include necessary imports and comments.
6. Only output the code, do not output any additional explanation.

Output Format

You must output only a Python code block (enclosed in triple backticks and python: ```` ```python ````), without any extra text.
The code block should contain necessary imports, optional auxiliary functions, and several `validate_*` functions.

Figure 31: Prompt template for constraint validation code generation (format & content).