SCOPEIF: IMPROVING SCOPE-AWARE PRECISE INSTRUCTION-FOLLOWING IN LARGE LANGUAGE MODELS VIA GRADED REWARD MODELING

Anonymous authors
Paper under double-blind review

ABSTRACT

Precise instruction-following is a fundamental ability of large language models (LLMs), requiring their outputs to strictly satisfy objective constraints in input instructions. In complex application scenarios, these constraints often possess diverse scopes that govern specific response segments rather than the entire output. However, existing optimization methods often neglect constraint scope during data construction and rely on binary per-constraint rewards, yielding limited data diversity and sparse supervision for complex constraints. To this end, we propose ScopeIF, a novel training framework for scope-aware precise instruction-following. We first introduce a unified schema that factorizes objective constraints into three decoupled dimensions: *Scope*, *Target*, and *Range*. Grounded in this schema, we construct SCOPEINSTRUCT, a large-scale instruction dataset with diverse scope-aware constraints, and combine tool-grounded verification with graded reward modeling to quantify the violation degree of each constraint, providing dense supervision for policy optimization. Extensive experiments demonstrate that ScopeIF consistently outperforms existing methods, particularly on complex scope-aware constraints, while preserving general capabilities. Notably, it enables optimized Qwen3-4B and 8B models to rival or surpass strong frontier models such as Gemini-2.5-Pro and DeepSeek-V3.2, establishing an effective paradigm for advancing scope-aware instruction-following. Our code and data are available at https://anonymous.4open.science/r/ScopeIF.

1 INTRODUCTION

Large language models (LLMs) have demonstrated remarkable capabilities across various NLP tasks (Zhao et al., 2023). Among these capabilities, precise instruction following is a foundational prerequisite for practical applications, requiring LLMs to accurately satisfy objective output constraints that specify length, format, and content (Pyatkin et al., 2026). Strict adherence to these constraints ensures model reliability (Huang et al., 2024) and is particularly vital in agentic and tool-assisted workflows that demand rigorous compliance with predefined schemas. In such scenarios, even minor deviations can cause parsing failures and downstream system breakdowns (Ye et al., 2026).

While current LLMs demonstrate proficiency in handling simple constraints applied to the entire response (Yang et al., 2025; Guo et al., 2025), they still struggle with scope-aware constraints, which govern specific segments of a response (Pyatkin et al., 2026; Mao & Chen, 2026), as shown in Figure 1. These constraints require fine-grained control and meticulous planning, posing a greater challenge to LLMs. Given the prevalence of such constraints in real-world complex instructions (Mao & Chen, 2026), there is a critical need for effective approaches to enhance this capability.

Recent advances in reinforcement learning (RL) provide a promising strategy to improve instruction-following (Peng et al., 2025a; Liu et al., 2025; Qin et al., 2026). They typically synthesize complex instructions with multiple constraints and optimize models with verifiable rewards. Despite these efforts, they still face two critical limitations when handling scope-aware constraints. First, for **data construction**, current methods primarily synthesize constraints applied to the entire response, neglecting the modeling of constraint scope (He et al., 2024; Zhang et al., 2025b; Cheng et al., 2025; Peng et al., 2025a). Although some benchmarks attempt to include scope-aware constraints (Pyatkin

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

et al., 2026; Mao & Chen, 2026), they remain restricted to a narrow set of hand-crafted, code-verifiable constraints, severely hindering diversity. Second, for **reward modeling**, existing methods typically conduct binary verification on individual constraints and aggregate their scores to derive the overall reward (Liu et al., 2025; Zhang et al., 2025a; Qin et al., 2026). However, as many scope-aware constraints govern multiple response segments (e.g., every sentence) or entail complex inter-segment dependencies (e.g., progressively increasing the word count across paragraphs) (Pyatkin et al., 2026), LLMs often struggle to satisfy them perfectly, even after many attempts. Consequently, the binary reward tends to become highly sparse and ineffective for guiding optimization.

To bridge this gap, we propose ScopeIF, a novel training framework for scope-aware precise instruction following. Our core innovation is a unified schema that formalizes objective constraints into three decoupled dimensions: *Scope* (governed segment), *Target* (constrained object), and *Range* (permissible bounds), which enables the modeling of constraint scopes and the computation of constraint violation degrees, directly addressing the aforementioned bottlenecks in data diversity and reward sparsity. Grounded in this schema, ScopeIF is driven by three synergistic components. First, **factorized constraint synthesis**. To collect diverse scope-aware instructions, we flexibly combine the three dimensions to generate atomic constraints. LLM-based crossover is then applied to merge them into complex instructions with multiple constraints. Second, **tool-grounded constraint verification**. To precisely count elements and isolate relevant response segments for complex scope-aware constraint verification, we empower judge models to dynamically design and execute tools based on response content, and then leverage the execution outcomes to derive fine-grained statistics of each *Target*. This method blends the flexibility of LLM judges with the precision of rules. Finally, **graded reward modeling**. To obtain denser rewards for effective model optimization, we quantify the violation degree of each constraint by comparing the statistics of each *Target* against the defined *Range* to derive a per-constraint continuous graded reward. This reward is then hierarchically aggregated with the binary reward to guide RL training. Our main contributions are summarized as follows:

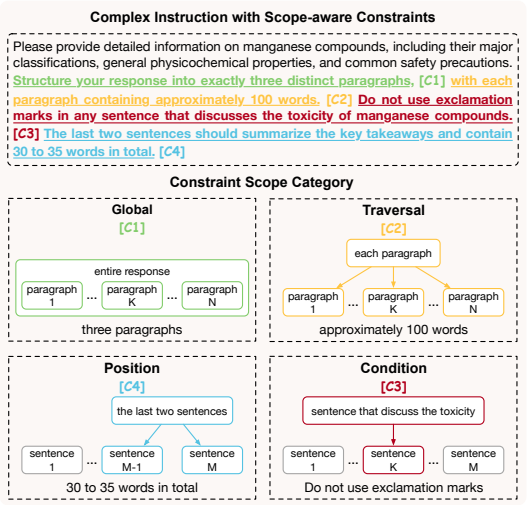


Figure 1: An example of instruction with scope-aware constraints that govern specific response segments rather than the entire response.

- We introduce a unified schema that factorizes objective constraints into three decoupled dimensions, enabling both diverse scope-aware constraint synthesis and constraint violation degree computation. Building on this, we construct ScopeInstruct, a large-scale dataset containing 17,968 complex instructions with diverse scope-aware constraints.
- We develop ScopeIF, a novel RL training framework for scope-aware precise instruction-following that combines tool-grounded constraint verification with a graded reward modeling mechanism to provide reliable and dense supervision for effective RL optimization.
- We validate the effectiveness of ScopeIF across multiple policy models and benchmarks, yielding consistently better performance compared to existing frameworks and reward modeling paradigms, particularly on complex scope-aware constraints.

2 Related Work

Precise Instruction-Following. As LLMs are increasingly deployed in complex applications, the ability to precisely follow instructions with multiple objective constraints has become essential to their practical utility (Liu et al., 2023; Lou et al., 2024), driving extensive research to evaluate and enhance this capability from various perspectives, such as output length (Jie et al., 2024; Zhang et al., 2026b;a) and format (Xia et al., 2024; Yao et al., 2025; Wang et al., 2025). These works typically construct verifiable constraints and use rule-based feedback for automatic evaluation or

model training. Building on this, IFEval (Zhou et al., 2023) extends precise instruction-following to 25 constraint categories, improving evaluation comprehensiveness. Furthermore, IFBench (Pyatkin et al., 2026) and IFHierBench (Mao & Chen, 2026) introduce hierarchical structures and scope modeling within constraints, posing greater challenges to LLMs. Nevertheless, these works confine precise instruction-following to a narrow set of hand-crafted, code-verifiable constraints. In contrast, our factorized synthesis method can automatically generate diverse scope-aware constraints at scale.

Instruction-Following Optimization. Recent algorithmic innovations have significantly advanced the instruction-following capabilities of LLMs, which evolve from supervised fine-tuning (Sun et al., 2024; Qi et al., 2025) to preference optimization (Zhang et al., 2025b; Cheng et al., 2025) and RL (Liu et al., 2025; Zhang et al., 2025a; Wen et al., 2026a). These approaches typically synthesize complex instructions with multiple constraints, conduct binary verification on each constraint, and aggregate their scores to guide model optimization. Specifically, RECAST (Liu et al., 2025) and VerIF (Peng et al., 2025a) categorize constraints into soft and hard types, leveraging LLM-based and rule-based evaluation, respectively, to derive rewards for RL training. However, these methods neglect constraint scope modeling during data construction and rely on sparse binary reward for each constraint that cannot distinguish different degrees of constraint violation. ScopeIF addresses these limitations with a unified constraint schema that enables scope-aware data synthesis and a graded reward mechanism to capture partial constraint compliance, yielding denser supervision signals.

3 PRELIMINARIES

In this section, we define the precise instruction-following task and its standard evaluation metrics.

Task Definition. Our goal is to enhance the precise instruction-following capabilities of LLMs. Formally, given an instruction x comprising a set of constraints $C = \{c_1, c_2, \ldots, c_n\}$, an LLM is considered to follow the instruction if its response y satisfies all constraints in C (Zhou et al., 2023). Precise instruction-following focuses on objective, deterministic constraints requiring no subjective semantic evaluation (Pyatkin et al., 2026), which constitute the vast majority of output constraints in real-world developer prompts (Mao & Chen, 2026). Furthermore, beyond applying to the entire response, scope-aware constraints govern specific segments of the response (e.g., particular paragraphs or sentences), which is prevalent in real-world complex instructions (Mao & Chen, 2026).

Evaluation Metrics. For each constraint, let the binary function $I(x, y, c_i) \in \{0, 1\}$ to denote whether a response y satisfies the constraint c_i. The overall instruction-following quality of a response is typically evaluated using the following metrics (Zhang et al., 2025a; Peng et al., 2025a): **(1) Instruction-Level Accuracy (ILA)** reflects strict adherence to the entire instruction. A response y is considered correct if and only if it satisfies every constraint of the instruction x, which is computed as: $\text{ILA}(x, y, C) = \prod_{c_i \in C} I(x, y, c_i)$. **(2) Constraint-Level Accuracy (CLA)** measures the ability to follow individual atomic constraints, calculated as the percentage of satisfied constraints: $\text{CLA}(x, y, C) = \frac{1}{|C|} \sum_{c_i \in C} I(x, y, c_i)$. While ILA and CLA are both common reward modeling approaches for RL training, they rely on binary signals for individual constraints without measuring their violation degrees, leading to a sparse reward problem when facing complex constraints.

4 DATA CONSTRUCTION

In this section, we introduce a factorized constraint synthesis method and construct SCOPEINSTRUCT, a large-scale dataset of complex instructions with diverse scope-aware constraints. We first formalize objective constraints through a unified schema comprising three dimensions: *Scope*, *Target*, and *Range*. Building on this, we generate diverse atomic constraints via dimension recombination and employ LLM-based crossover to synthesize instructions with multiple constraints.

4.1 UNIFIED CONSTRAINT SCHEMA

Existing constraint synthesis methods treat constraints as indivisible templates, confining them to a manually designed inventory and precluding a unified metric for quantifying violation degrees. We observe that objective constraints inherently share a unified schema: they dictate the permissible

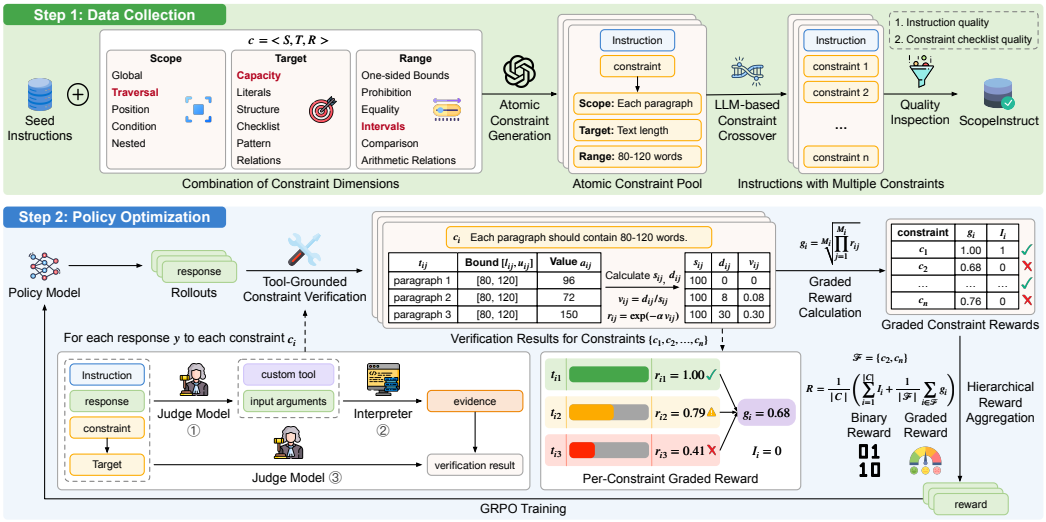


Figure 2: Overall framework of ScopeIF. *Top:* Complex instruction collection via factorized constraint synthesis. *Bottom:* Policy model optimization via graded reward modeling.

bounds of specific measurable objects within designated response segments. Driven by this observation, we factorize each constraint c_i into three decoupled dimensions: *Scope*, *Target*, and *Range*. This decomposition enables diverse constraint synthesis via dimension recombination, while providing a unified basis for quantifying violation degrees. Formally, we define this schema as follows:

$$c_i = \langle \mathcal{S}_i, \mathcal{T}_i, \mathcal{R}_i \rangle ,$$
$$\mathcal{S}_i(y) = \{\sigma_{ij}\}_{j=1}^{M_i}, \qquad \mathcal{T}_i(y) = \{t_{ij}\}_{j=1}^{M_i}, \qquad \mathcal{R}_i(y) = \{[l_{ij}, u_{ij}]\}_{j=1}^{M_i}. \tag{1}$$

Here, $\mathcal{S}_i$ denotes *Scope* selector, which selects M_i governed segments $\{\sigma_{ij}\}$ from response y ($M_i > 1$ accommodates multi-segment scopes, such as every sentence or paragraph). *Target* $\mathcal{T}_i$ yields the measurable object t_{ij} for each segment, and *Range* $\mathcal{R}_i$ assigns its permissible bounds $[l_{ij}, u_{ij}]$. By systematically analyzing instructions from real-world applications and existing benchmarks, we establish a comprehensive taxonomy for these three dimensions. More details are in Appendix A.

Scope selector $\mathcal{S}_i$ isolates governed segments by applying a selection operation to response units, spanning the entire response, structural blocks and elements, paragraphs, sentences, words, and characters. The selector falls into four categories: **(1) Global** covers the entire response, **(2) Traversal** selects every unit of a given kind (e.g., every paragraph), **(3) Position** targets units at specific locations (e.g., the beginning), and **(4) Condition** filters units through predicates (e.g., sentences containing a specific keyword). These operations can be recursively composed to express nested scopes (e.g., the final sentence of every paragraph).

Target $\mathcal{T}_i$ specifies objects measured within the selected *Scope*. We organize 15 subcategories into six major categories: **(1) Capacity** covers text length and unit counts (e.g., sentences or paragraphs), **(2) Literals** tracks specific strings and literal classes (e.g., keywords or punctuation), **(3) Structure** captures containers, members, and nesting depth in structured formats (e.g., JSON or Markdown), **(4) Checklist** assesses mandatory elements and valid values (e.g., specific JSON keys), **(5) Pattern** verifies conformity to boundary, skeleton, and surface templates (e.g., enclosed quotes), and **(6) Relations** characterizes multi-object dependencies (e.g., alphabetical ordering or acrostic poetry).

Range $\mathcal{R}_i$ specifies the admissible values of *Target* through four absolute and two relative categories. The absolute categories comprise: **(1) One-Sided Bounds** imposes a minimum or maximum, **(2) Prohibition** fixes the value at zero (e.g., forbidding keywords), **(3) Equality** prescribes an exact value (e.g., exactly five sentences), and **(4) Intervals** defines a closed numeric range (e.g., 80 to 120 words). The relative categories derive admissible values from another response measurement: **(5) Comparison** imposes inequalities (e.g., increasing word counts per sentence), and **(6) Arithmetic Relations** enforces precise equations. Crucially, this schema accommodates qualitative constraints (e.g., valid JSON format) by formulating compliance as an **Equality** *Range* of one.

4.2 Instruction Generation Pipeline

Atomic Constraint Generation. To gather a diverse and representative pool of atomic constraints, we enumerate all compatible combinations of *Scope*, *Target*, and *Range* categories. For each combination, we chain 10 generation iterations. In each iteration, we prompt a strong LLM to generate a batch of five distinct instructions from a seed, each incorporating an atomic constraint instantiated from that combination. The seed instruction grounds synthesis in realistic contexts, while batch generation improves efficiency. To avoid repetition, previously generated constraints for the same combination are provided as context history, guiding the LLM to elicit novel realizations.

LLM-Based Constraint Crossover. After establishing the constraint pool, we perform constraint crossover to synthesize multi-constraint instructions of varying complexity. Guided by a predefined target $k \in \{2, 4, 6, 8\}$, we randomly sample 16 candidate constraints per iteration and prompt a strong LLM to construct five distinct instructions from a seed, each integrating at least k of these constraints. During integration, the LLM is permitted to modify both the seed and the sampled constraints to ensure fluency and coherence. To facilitate downstream verification, the LLM is also required to output a constraint list for each instruction, alongside *Target* for each constraint.

Data Curation and Quality Control. We derive seed instructions from HIR-16k (Zhang et al., 2025a) and MulDimIF (Ye et al., 2026), applying the above process to construct instructions with single or multiple scope-aware constraints. Then, we employ strong LLMs to assess the quality of instructions and verify their constraint lists, discarding any instances with ambiguous intents, semantic conflicts, unreasonable requests, or inaccurate constraint lists. Finally, we sample the retained instructions across varying constraint counts to ensure diverse instruction complexity. The resulting ScopeInstruct comprises 16,968 training and 1,000 test instructions, each containing 1 to 12 constraints (4.83 on average). Detailed data statistics of ScopeInstruct are in Appendix C.

5 Methodology

In this section, we introduce ScopeIF, an RL training framework for scope-aware precise instruction-following. We first employ a tool-grounded verification pipeline to derive granular response statistics for each constraint. A graded reward modeling mechanism is then applied to quantify constraint violations and provide fine-grained reward signals. Without loss of generality, we adopt the Group Relative Policy Optimization (GRPO) (Shao et al., 2024) algorithm for RL training.

5.1 Tool-Grounded Constraint Verification

Verifying scope-aware constraints is challenging, as it requires both precisely isolating relevant response segments and counting elements. For example, verifying *Generate 10 titles, each containing 8 words* requires extracting all titles in the response before calculating length, yet titles can be delimited in diverse ways. Rule-based verifiers offer precision but are limited to rigid constraints and fixed delimitation, whereas LLM judges are flexible but struggle with exact counting. To bridge this gap, we empower judge models to dynamically design and execute verification tools conditioned on the response rather than statically pre-defined for constraint alone (Peng et al., 2025b; Wen et al., 2024), thereby accommodating response-specific delimitation. Specifically, for instruction x, its constraint c_i with *Target* $\mathcal{T}_i = \{t_{ij}\}_{j=1}^{M_i}$, and response y, our verification pipeline operates as follows:

Tool Design and Execution. To compute any auxiliary evidence required for deriving the statistics of $\mathcal{T}_i$, the judge model J first designs a custom Python tool f_i (if needed) along with its input arguments $\mathbf{z}_i$ tailored for y:

$$(f_i, \mathbf{z}_i) = J(x, c_i, \mathcal{T}_i, y) \tag{2}$$

An interpreter then executes the generated code to yield this evidence $\mathbf{e}_i$ for accurate verification:

$$\mathbf{e}_i = f_i(\mathbf{z}_i) \tag{3}$$

Tool-Grounded Judgment. Grounded in this evidence and response content, the judge model then generates granular statistics $\mathcal{V}_i(y)$ for $\mathcal{T}_i$, which serve as the verification result for constraint c_i:

$$\mathcal{V}_i(y) = J(x, c_i, \mathcal{T}_i, y, \mathbf{e}_i) \tag{4}$$

Specifically, $\mathcal{V}_i(y)$ is a set of quantitative records $\{([l_{ij}, u_{ij}], a_{ij})\}_{j=1}^{M_i}$, each pairing the permissible bound $[l_{ij}, u_{ij}]$ of object t_{ij} with its measured value a_{ij}. These records yield the binary satisfaction indicator $I_i(y) = \mathbb{1}(\forall j, a_{ij} \in [l_{ij}, u_{ij}])$ and enable the subsequent graded reward computation.

5.2 Graded Constraint Reward Modeling

To alleviate the sparsity of binary rewards for complex constraints, we quantify each constraint's violation degree from its verification statistics and hierarchically aggregate the resulting graded reward with the binary satisfaction signal of each constraint into a denser response-level reward.

Graded Reward Calculation. For each constraint c_i, we first quantify the violation degree of each object $t_{ij} \in \mathcal{T}_i$. To normalize violations across heterogeneous scales, we compute a normalized violation metric $v_{ij} = d_{ij}/s_{ij}$, where d_{ij} is the absolute deviation of the measured value a_{ij} from the permissible bound $[l_{ij}, u_{ij}]$ (with $d_{ij} = 0$ if $a_{ij} \in [l_{ij}, u_{ij}]$), and s_{ij} denotes its characteristic range magnitude. The graded reward of t_{ij} is then computed via exponential decay:

$$r_{ij} = \exp(-\alpha v_{ij}) \tag{5}$$

where α controls the penalty severity, and $r_{ij} = 1$ indicates full satisfaction. We then derive the graded reward g_i of c_i as the geometric mean of rewards across all objects in $\mathcal{T}_i$:

$$g_i = \sqrt[M_i]{\prod_{j=1}^{M_i} r_{ij}} \tag{6}$$

Hierarchical Final Reward Aggregation. Directly averaging graded rewards $\{g_i\}$ risks reward misalignment, where accumulating partial scores may outweigh fully satisfying constraints. To address this problem, we hierarchically aggregate the binary satisfaction indicator $I_i(y)$ with the mean graded reward of the violated constraint subset $\mathcal{F} = \{i \mid I_i(y) = 0\}$ (taken as 0 if $\mathcal{F} = \varnothing$):

$$R(x, y) = \frac{1}{|C|}\left(\underbrace{\sum_{i=1}^{|C|} I_i(y)}_{\text{binary reward}} + \underbrace{\frac{1}{|\mathcal{F}|}\sum_{i \in \mathcal{F}} g_i}_{\text{graded reward}}\right) \tag{7}$$

This formulation bounds $R(x, y) \in [m/|C|, (m+1)/|C|)$ when m constraints are fully satisfied. Thus, satisfying more constraints guarantees a discrete reward leap, while the graded reward distinguishes degrees of partial compliance and provides dense supervision for failures. Finally, this reward is integrated with a format reward for GRPO training. More details are in Appendix D.

6 Experiments

6.1 Experimental Setup

Evaluation Benchmarks. Alongside our test set, we evaluate precise instruction-following on three public benchmarks, including IFEval (Zhou et al., 2023), IFBench (Pyatkin et al., 2026), and IFHierBench (Mao & Chen, 2026). IFEval focuses on simple global objective constraints, while IFBench and IFHierBench focus on more complex scope-aware constraints. Additionally, we evaluate our method across three general benchmarks, including MATH-500 (Lightman et al., 2024), GPQA (Rein et al., 2023), and MMLU-Pro (Wang et al., 2024). Following ImpRIF (Yang et al., 2026), our test set uses two metrics: CSR (Constraint Success Rate) for the proportion of individual constraints satisfied, and ISR (Instruction Success Rate) for the proportion of samples where all constraints are satisfied. Details of evaluation metrics are presented in Appendix E.1.

Baselines. We choose Qwen3-4B (Yang et al., 2025) and Qwen3-8B as initial policy models. The baselines encompass three categories: 1) Off-the-shelf LLMs, including the original policy models alongside several strong open-source and proprietary LLMs; 2) Alternative optimization methods based on ScopeInstruct and our constraint verification method, including Direct Preference Optimization (DPO) (Rafailov et al., 2023), Reinforcement Learning (RL) with Instruction-Level

Table 1: Experimental results (%) of different optimization methods on instruction-following benchmarks. P and I stand for prompt-level and instruction-level metrics, respectively. L and S denote loose and strict evaluations, respectively. HRA means hierarchical reward aggregation. The highest result for each backbone model is **bolded**, and the overall highest is underlined.

Model	ScopeIF-Test		IFEval				IFBench				IFHierBench		Avg.
	ISR	CSR	P-S	P-L	I-S	I-L	P-S	P-L	I-S	I-L	P-S	I-S	
Frontier Models													
Seed-2.0-Pro	21.8	69.5	88.2	91.7	92.3	94.6	49.3	56.0	51.5	57.6	22.8	52.8	57.2
Gemini-2.5-Pro	15.8	63.7	90.0	93.0	92.8	95.2	40.0	47.7	43.0	50.9	20.8	46.7	52.9
DeepSeek-V3.2	9.1	50.8	87.4	91.5	91.2	94.0	38.3	43.3	42.2	47.1	14.3	33.7	46.9
GPT-4.1	8.6	53.1	87.8	91.3	91.5	94.4	36.7	41.3	41.3	46.2	18.2	44.1	48.7
QwQ-32B	8.5	48.7	82.4	87.6	88.1	91.6	34.7	39.7	38.4	43.3	17.2	37.2	45.6
Our Models													
Qwen3-4B	5.2	40.6	81.7	85.6	86.8	89.6	29.7	34.3	32.8	38.1	15.0	25.0	40.6
+ DPO	5.6	43.5	86.1	88.7	90.6	92.6	32.0	36.7	34.6	40.7	15.7	27.6	42.9
+ RL-ILA	10.5	50.5	85.4	87.4	89.9	91.5	47.7	50.0	50.9	53.8	17.3	22.0	47.3
+ RL-CLA	12.8	55.2	85.8	89.6	90.4	92.9	48.0	51.3	50.3	53.5	18.8	32.6	50.0
+ ScopeIF (Ours)	**15.3**	**60.9**	**88.2**	**90.9**	**92.2**	**94.1**	**49.7**	**53.3**	**52.0**	**56.1**	18.8	**39.4**	**52.8**
- *wo HRA*	14.1	58.6	87.2	90.8	91.6	93.9	48.0	50.7	50.0	52.6	**20.7**	36.7	51.6
Qwen3-8B	7.2	43.8	85.0	88.4	89.3	91.6	31.0	33.7	34.9	38.4	15.0	25.5	42.2
+ DPO	6.8	45.2	85.6	89.3	90.0	92.8	33.7	39.3	36.3	42.4	15.8	26.7	43.7
+ RL-ILA	14.2	55.1	87.1	88.5	91.1	92.2	45.7	50.0	47.0	51.5	18.2	31.3	49.4
+ RL-CLA	17.7	61.9	88.4	91.1	92.2	94.0	51.0	55.0	52.9	56.7	20.5	35.6	53.3
+ ScopeIF (Ours)	**19.6**	**64.9**	**90.4**	**92.2**	**93.3**	**94.6**	**53.7**	**57.0**	**57.0**	**59.9**	**21.7**	36.6	**55.2**
- *wo HRA*	16.0	62.2	88.4	90.8	91.8	93.5	50.3	55.0	52.9	57.3	21.3	**38.6**	53.5

Table 2: Experimental results (%) of different instruction-following improving frameworks. To ensure fairness in the amount of training data, MulDimIF used twice the number of training epochs.

Model	Method	#Data	ScopeIF-Test		IFEval				IFBench				IFHierBench		Avg.
			ISR	CSR	P-S	P-L	I-S	I-L	P-S	P-L	I-S	I-L	P-S	I-S	
Qwen3-4B	HIR	16k	6.7	44.2	87.2	**91.0**	90.4	92.9	41.7	44.7	44.2	47.7	16.0	25.4	45.3
	VerIF	22k	7.9	47.7	86.5	89.8	91.0	93.3	42.3	48.3	45.3	50.9	16.5	30.9	47.1
	MulDimIF	8k	6.0	42.6	85.6	88.5	90.4	92.3	39.3	43.0	41.6	46.5	14.7	21.6	43.6
	ScopeIF (Ours)	16k	**15.3**	**60.9**	**88.2**	90.9	**92.2**	**94.1**	**49.7**	**53.3**	**52.0**	**56.1**	**18.8**	**39.4**	**52.8**
Qwen3-8B	HIR	16k	8.6	48.9	89.5	92.1	92.7	94.6	39.3	44.3	41.9	46.8	17.3	30.7	47.0
	VerIF	22k	10.0	46.9	87.1	91.1	91.1	94.0	42.0	47.3	44.8	50.3	16.7	27.9	46.9
	MulDimIF	8k	6.4	43.5	86.7	89.5	91.2	93.0	34.7	40.7	38.1	43.3	15.8	20.5	43.1
	ScopeIF (Ours)	16k	**19.6**	**64.9**	**90.4**	**92.2**	**93.3**	**94.6**	**53.7**	**57.0**	**57.0**	**59.9**	**21.7**	**36.6**	**55.2**

Accuracy as the reward (Peng et al., 2025a), and RL with Constraint-Level Accuracy as the reward (Qin et al., 2026; Liu et al., 2025); and 3) Existing instruction-following improving frameworks, including HIR (Zhang et al., 2025a), VerIF (Peng et al., 2025a), and MulDimIF (Ye et al., 2026).

Implementation Details. For data construction, we use Seed-2.0-Pro (Seed Team, 2026) for constraint generation and crossover, alongside GPT-OSS-120B[1] for instruction quality control. For reward calculation, the penalty severity α is set to 3. For GRPO training, we use the verl (Sheng et al., 2025) framework, performing 8 rollouts per instruction with a batch size of 32. Across all settings, GPT-OSS-120B serves as the judge model, with its reasoning effort configured to medium. Greedy search is used for evaluation to ensure reproducibility. More details are in Appendix E.2.

6.2 Main Results

Tables 1 and 2 present the performance of different optimization methods and instruction-following frameworks. **Firstly**, ScopeIF significantly enhances scope-aware instruction-following capabilities across different model scales. With average relative improvements of 60.7% and 44.7% on IFBench and IFHierBench, the optimized Qwen3-4B and 8B models can rival or even surpass strong frontier models such as Gemini-2.5-Pro and DeepSeek-V3.2. **Secondly**, ScopeIF outperforms all alternative optimization methods. RL approaches substantially beat DPO, and their effectiveness strongly correlates with reward granularity. Average performance improves progressively as rewards transition

[1] https://huggingface.co/openai/gpt-oss-120b

Table 3: Performance (%) of ScopeIF on general benchmarks.

Model	MATH-500	GPQA	MMLU-Pro
Qwen3-4B	87.8	54.0	70.7
+ ScopeIF (Ours)	87.6 ↓0.2	53.0 ↓1.0	70.0 ↓0.7
Qwen3-8B	89.2	59.6	74.8
+ ScopeIF (Ours)	88.6 ↓0.6	60.1 ↑0.5	75.2 ↑0.4

Table 4: Pointwise accuracy (Acc.) and pairwise agreement (Agr.) of judge models on our test set and Kendall correlation τ on IF-RewardBench under the constraint assessment setting.

Judge Model	Method	ScopeIF-Test		IF-RewardBench
		Acc.	Agr.	τ
Qwen3-32B	LLM	0.820	0.606	0.351
	Rule	0.799	0.668	0.372
	TGV (Ours)	**0.881**	**0.734**	**0.461**
	- *w/ Static Tool*	0.864	0.694	0.457
QwQ-32B	LLM	0.867	0.712	0.494
	Rule	0.841	0.726	0.419
	TGV (Ours)	**0.902**	**0.792**	**0.532**
	- *w/ Static Tool*	0.894	0.776	0.513
GPT-OSS-120B	LLM	0.923	0.810	0.512
	Rule	0.860	0.738	0.388
	TGV (Ours)	**0.935**	**0.842**	**0.558**
	- *w/ Static Tool*	0.922	0.804	0.548

Table 5: Constraint diversity of different training datasets for instruction-following.

Dataset	Self-BLEU-4 ↓	Distinct-2 ↑	Entropy-4 ↑
HIR	0.931	0.067	10.3
VerIF	0.918	0.068	11.0
MulDimIF	0.961	0.059	7.8
ScopeInstruct	**0.834**	**0.133**	**12.8**

from ILA to CLA and peaks with ScopeIF, which validates the value of our fine-grained graded constraint reward modeling mechanism. We also conduct an ablation study that removes hierarchical reward aggregation by directly averaging graded rewards. Results in Table 1 show performance drops on most metrics, highlighting the importance of hierarchical aggregation in encouraging full constraint compliance. **Thirdly**, ScopeIF consistently surpasses existing instruction-following improving frameworks, including HIR, VerIF, and MulDimIF. While the performance gap is marginal on IFEval's simple constraints, ScopeIF establishes a substantial lead on complex scope-aware constraints across the other three benchmarks. This highlights the oversight of scope-aware constraints in prior methods and the importance of our unified schema for constraint scope modeling. **Finally**, as shown in Table 3, ScopeIF preserves the general performance of LLMs with negligible degradation.

6.3 Analysis

Performance on Different Constraint Scopes. To dissect the ability of LLMs to follow constraints with specific scopes, we report the CSR of each *Scope* category on our test set in Figure 3. **Firstly**, scope complexity poses a growing challenge to LLMs. While base models perform best on Global, their performance drops markedly on single-layer scope selectors (i.e., Traversal, Position, and Condition), and reaches its lowest point on recursively composed Nested scopes. This trend highlights the inherent difficulty of scope-aware instruction-following and aligns with our motivation. **Secondly**, the advantage of ScopeIF over binary-reward baselines widens as scope complexity increases, which is marginal on Global yet grows steadily toward single-layer and nested scopes. Complex scopes render binary rewards increasingly sparse, whereas our graded reward still quantifies partial violations to yield denser learning signals, confirming its particular value for complex scope-aware constraints. We also analyze the performance of LLMs on different constraint *Target* and *Range* categories in Appendix E.3.

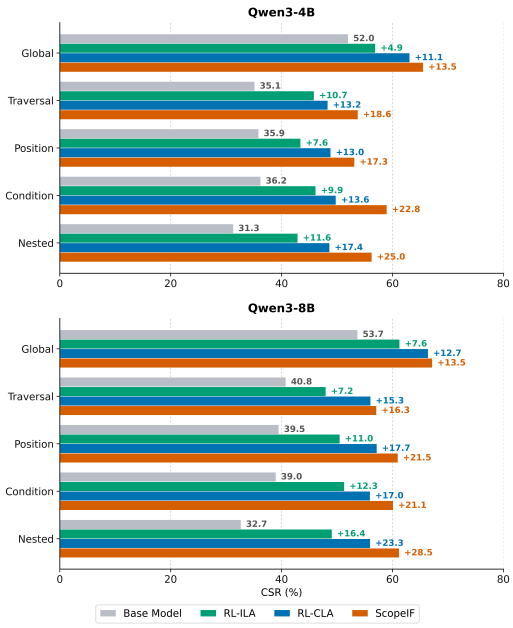


Figure 3: Performance of constraints with different scope categories on our test set.

Constraint Verification Methods. To validate the reliability of our verification methods, we construct a meta-evaluation set of 500 instructions sampled from our test set, each paired with responses from 2 of 6 LLMs of varying capability. Two annotators independently judge compliance for each constraint, with a third inspector cross-validating and resolving discrepancies by consensus. The

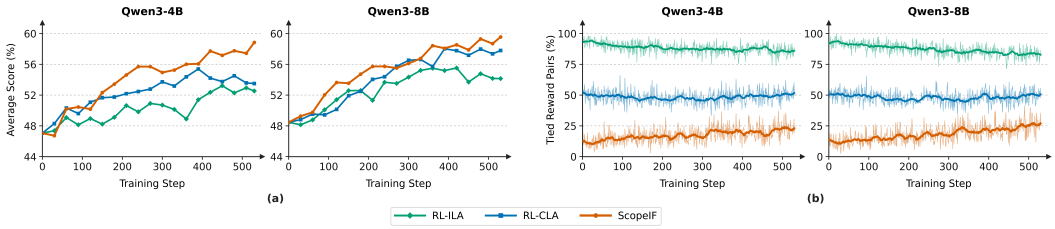


Figure 4: Average performance of IFEval, IFBench, and IFHierBench (a), and the proportion of rollout response pairs with identical rewards (b) during ScopeIF training.

resulting dataset comprises 5,350 constraints, of which 1,843 are labeled as violated. Under three judge models, we compare our Tool-Grounded Verification (TGV) method with pure LLM judges (LLM) and rule-based verifiers (Rule) using two metrics: pointwise accuracy, the per-constraint agreement with human labels; and pairwise agreement, the fraction of response pairs for which the judge and humans agree on which response satisfies more constraints. As shown in Table 4, TGV combines the flexibility of LLM judges with the precision of rule-based verifiers through dynamic tool generation, substantially outperforming both baselines across all judge models. Moreover, generating the verification tool without conditioning on the response (w/ Static Tool) causes a consistent drop, confirming the benefit of tailoring custom tools to each response. Beyond our own test set, we further evaluate these methods under the constraint assessment setting of IF-RewardBench (Wen et al., 2026b), which measures the ability of verifiers to rank the instruction-following quality of multiple responses. Here, TGV also achieves the best performance, confirming its generalization to diverse constraint types. More details are in Appendix E.4.

Constraint Diversity. To validate the effectiveness of our factorized constraint synthesis method in improving constraint diversity, we compare the constraint-level diversity of SCOPEINSTRUCT with three widely used baseline training datasets. Three automatic metrics are utilized: (1) Self-BLEU-4 (Zhu et al., 2018) measures inter-constraint similarity by treating each constraint as the hypothesis and the remaining constraints as references; (2) Distinct-2 (Li et al., 2016) calculates the proportion of unique bigrams among all bigram occurrences; and (3) Entropy-4 (Zhang et al., 2018) quantifies the diversity and distributional balance of four-grams using Shannon entropy. Table 5 shows that SCOPEINSTRUCT consistently exhibits greater diversity across these measures, supporting the benefit of factorizing and recombining different dimensions for diverse constraint synthesis.

Training Curves. To examine the training dynamics of different reward designs, we trace the average score on IFEval, IFBench, and IFHierBench throughout RL training in Figure 4 (a). **Firstly**, ScopeIF outperforms both binary-reward baselines throughout the training process, demonstrating a robust and consistent advantage. **Secondly**, this advantage is more pronounced on Qwen3-4B than on Qwen3-8B, as smaller models may be more prone to the sparse-reward problem and thus benefit more from the denser signals of our graded reward. **Thirdly**, the curves have not yet plateaued under our current budget, with ScopeIF still trending upward at the final step. This suggests that further scaling of training resources, such as more training steps or rollouts, could unlock additional gains. To quantify the density of different reward mechanisms, we also trace the proportion of response pairs receiving identical rewards within each GRPO rollout group throughout RL training in Figure 4 (b), as identical rewards provide no relative preference between the paired responses. This ratio drops sharply as reward granularity increases from ILA to CLA and ScopeIF, and the gap persists throughout training, showing that our graded reward substantially alleviates reward sparsity.

7 CONCLUSION

We propose ScopeIF, a novel RL training framework for scope-aware precise instruction-following. We first introduce a unified schema that factorizes objective constraints into three dimensions, *Scope*, *Target*, and *Range*, enabling diverse constraint synthesis and violation degree computation. Grounded in this, we construct SCOPEINSTRUCT, a large-scale dataset of instructions with scope-aware constraints, and combine tool-grounded verification with graded reward modeling for reliable, dense RL supervision. Extensive experiments show that ScopeIF consistently outperforms existing frameworks and reward modeling paradigms, especially on complex scope-aware constraints.

AI USE STATEMENT

In this work, we used generative AI tools to generate synthetic datasets and to clean and reformat these data. We have not used them for any other required-disclosure task. Additionally, we used generative AI tools to polish our figures and manuscript. We have reviewed all AI-assisted work, verifying the generated data through our quality-control pipeline and checking all AI-assisted figures and text ourselves. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

To ensure the reproducibility of our experiments, we have released the full implementation of our ScopeIF framework (https://anonymous.4open.science/r/ScopeIF), which includes full data and code for data collection and RL training. The prompts applied throughout this work are provided in Appendix B. Crucial hyperparameters for training our models and the baselines are documented in Appendix E.2.

REFERENCES

Jiale Cheng, Xiao Liu, Cunxiang Wang, Xiaotao Gu, Yida Lu, Dan Zhang, Yuxiao Dong, Jie Tang, Hongning Wang, and Minlie Huang. SPar: Self-play with tree-search refinement to improve instruction-following in large language models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=9chRqsPOGL.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638, 2025.

Qianyu He, Jie Zeng, Qianxi He, Jiaqing Liang, and Yanghua Xiao. From complex to simple: Enhancing multi-constraint complex instruction following ability of large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 10864–10882, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.637. URL https://aclanthology.org/2024.findings-emnlp.637/.

Zhenyu Hou, Yilin Niu, Zhengxiao Du, Xiaohan Zhang, Xiao Liu, Aohan Zeng, Qinkai Zheng, Minlie Huang, Hongning Wang, Jie Tang, et al. Chatglm-rlhf: Practices of aligning large language models with human feedback. *arXiv preprint arXiv:2404.00934*, 2024.

Xiaowei Huang, Wenjie Ruan, Wei Huang, Gaojie Jin, Yi Dong, Changshun Wu, Saddek Bensalem, Ronghui Mu, Yi Qi, Xingyu Zhao, et al. A survey of safety and trustworthiness of large language models through the lens of verification and validation. *Artificial Intelligence Review*, 57(7):175, 2024.

Renlong Jie, Xiaojun Meng, Lifeng Shang, Xin Jiang, and Qun Liu. Prompt-based length controlled generation with multiple control types. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 1067–1085, 2024.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, et al. T\" ulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*, 2024.

Jiwei Li, Michel Galley, Chris Brockett, Jianfeng Gao, and William B Dolan. A diversity-promoting objective function for neural conversation models. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*, pp. 110–119, 2016.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations*, volume 2024, pp. 39578–39601, 2024.

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

Yang Liu, Yuanshun Yao, Jean-Francois Ton, Xiaoying Zhang, Ruocheng Guo, Hao Cheng, Yegor Klochkov, Muhammad Faaiz Taufiq, and Hang Li. Trustworthy llms: a survey and guideline for evaluating large language models' alignment. *arXiv preprint arXiv:2308.05374*, 2023.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

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

Yuetian Mao and Chunyang Chen. Ifhierbench: Hierarchical instruction following for large language models. *arXiv preprint arXiv:2607.27912*, 2026.

Hao Peng, Yunjia Qi, Xiaozhi Wang, Bin Xu, Lei Hou, and Juanzi Li. VerIF: Verification engineering for reinforcement learning in instruction following. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 30324–30339, Suzhou, China, November 2025a. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.1542. URL https://aclanthology.org/2025.emnlp-main.1542/.

Hao Peng, Yunjia Qi, Xiaozhi Wang, Zijun Yao, Bin Xu, Lei Hou, and Juanzi Li. Agentic reward modeling: Integrating human preferences with verifiable correctness signals for reliable reward systems. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15934–15949, Vienna, Austria, July 2025b. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.775. URL https://aclanthology.org/2025.acl-long.775/.

Valentina Pyatkin, Saumya Malik, Victoria Graf, Hamish Ivison, Shengyi Huang, Pradeep Dasigi, Nathan Lambert, and Hanna Hajishirzi. Generalizing verifiable instruction following. *Advances in Neural Information Processing Systems*, 38, 2026.

Yunjia Qi, Hao Peng, Xiaozhi Wang, Bin Xu, Lei Hou, and Juanzi Li. Constraint back-translation improves complex instruction following of large language models. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management*, pp. 2388–2398, 2025.

Yulei Qin, Gang Li, Zongyi Li, Zihan Xu, Yuchen Shi, Zhekai Lin, Xiao Cui, Ke Li, and Xing Sun. Incentivizing reasoning for advanced instruction-following of large language models. *Advances in Neural Information Processing Systems*, 38:108337–108401, 2026.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, 2020.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 3505–3506, 2020.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

Seed Team. Seed 2.0 official launch, 2026. URL https://seed.bytedance.com/en/blog/seed-2-0-official-launch.

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

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pp. 1279–1297, 2025.

Haoran Sun, Lixin Liu, Junjie Li, Fengyu Wang, Baohua Dong, Ran Lin, and Ruohui Huang. Conifer: Improving complex constrained instruction-following ability of large language models. *arXiv preprint arXiv:2404.02823*, 2024.

Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, et al. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. *Advances in Neural Information Processing Systems*, 37:95266–95290, 2024.

Zhaoyang Wang, Jinqi Jiang, Huichi Zhou, Wenhao Zheng, Xuchao Zhang, Chetan Bansal, and Huaxiu Yao. Verifiable format control for large language model generations. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 3499–3513, 2025.

Bosi Wen, Pei Ke, Xiaotao Gu, Lindong Wu, Hao Huang, Jinfeng Zhou, Wenchuang Li, Binxin Hu, Wendy Gao, Jiaxin Xu, et al. Benchmarking complex instruction-following with multiple constraints composition. *Advances in Neural Information Processing Systems*, 37:137610–137645, 2024.

Bosi Wen, Yilin Niu, Cunxiang Wang, Pei Ke, Xiaoying Ling, Ying Zhang, Aohan Zeng, Hongning Wang, and Minlie Huang. IF-CRITIC: Towards a fine-grained LLM critic for instruction-following evaluation. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 25004–25029, San Diego, California, United States, July 2026a. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.1147. URL https://aclanthology.org/2026.acl-long.1147/.

Bosi Wen, Yilin Niu, Cunxiang Wang, Xiaoying Ling, Ying Zhang, Pei Ke, Hongning Wang, and Minlie Huang. IF-RewardBench: Benchmarking judge models for instruction-following evaluation. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 23816–23843, San Diego, California, United States, July 2026b. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.1092. URL https://aclanthology.org/2026.acl-long.1092/.

Congying Xia, Chen Xing, Jiangshu Du, Xinyi Yang, Yihao Feng, Ran Xu, Wenpeng Yin, and Caiming Xiong. Fofo: A benchmark to evaluate llms' format-following capability. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 680–699, 2024.

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

Yuancheng Yang, Lin Yang, Xu Wang, Chao Tong, and Haihua Yang. ImpRIF: Stronger implicit reasoning leads to better complex instruction following. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 38771–38796, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.1796. URL https://aclanthology.org/2026.acl-long.1796/.

Jiashu Yao, Heyan Huang, Zeming Liu, Haoyu Wen, Wei Su, Boao Qian, and Yuhang Guo. Reff: Reinforcing format faithfulness in language models across varied tasks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 25660–25668, 2025.

Junjie Ye, Caishuang Huang, Zhuohan Chen, Wenjie Fu, Chenyuan Yang, Leyi Yang, Yilong Wu, Peng Wang, Meng Zhou, Xiaolong Yang, et al. Muldimif: A multi-dimensional constraint framework for evaluating and improving instruction following in large language models. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 2078–2104, 2026.

Kongcheng Zhang, Qi Yao, Shunyu Liu, Wenjian Zhang, Min Cen, Yang Zhou, Wenkai Fang, Yiru Zhao, Baisheng Lai, and Mingli Song. Replay failures as successes: Sample-efficient reinforcement learning for instruction following. *arXiv preprint arXiv:2512.23457*, 2025a.

Wei Zhang, Lintong Du, Yuanhe Zhang, Zhenhong Zhou, Kun Wang, Li Sun, and Sen Su. Larft: Closing the cognition-action gap for length instruction following in large language models. *arXiv preprint arXiv:2603.19255*, 2026a.

Wei Zhang, Zhenhong Zhou, Kun Wang, Junfeng Fang, Rongwu Xu, Yuanhe Zhang, Rui Wang, Ge Zhang, Xinfeng Li, Li Sun, et al. Lifebench: Evaluating length instruction following in large language models. *Advances in Neural Information Processing Systems*, 38, 2026b.

Xinghua Zhang, Haiyang Yu, Cheng Fu, Fei Huang, and Yongbin Li. IOPO: Empowering LLMs with complex instruction following via input-output preference optimization. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 22185–22200, Vienna, Austria, July 2025b. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.1079. URL https://aclanthology.org/2025.acl-long.1079/.

Yizhe Zhang, Michel Galley, Jianfeng Gao, Zhe Gan, Xiujun Li, Chris Brockett, and Bill Dolan. Generating informative and diverse conversational responses via adversarial information maximization. *Advances in Neural Information Processing Systems*, 31, 2018.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2), 2023.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 3: system demonstrations)*, pp. 400–410, 2024.

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

Yaoming Zhu, Sidi Lu, Lei Zheng, Jiaxian Guo, Weinan Zhang, Jun Wang, and Yong Yu. Texygen: A benchmarking platform for text generation models. In *The 41st international ACM SIGIR conference on research & development in information retrieval*, pp. 1097–1100, 2018.

A Details of Constraint Schema

In this section, we provide the detailed taxonomy of the three dimensions defined in § 4.1: *Scope*, *Target*, and *Range*.

A.1 Scope

The *Scope* selector $\mathcal{S}_i$ isolates governed segments by applying a selection operation to response units. Therefore, the *Scope* of a constraint is instantiated by some response units and a selection operation.

Response Units. The selector operates over response units at the following granularities:

1. **Entire Response.** The complete response is treated as a single governed segment.
2. **Structural Blocks.** Complete structural regions, such as sections, lists, tables, and JSON, Markdown, or LaTeX blocks.
3. **Structural Elements.** Constituent elements within structural blocks, such as list items, table rows or cells, JSON key-value pairs, and Markdown headings.
4. **Paragraphs.** Individual paragraphs in the response.
5. **Sentences.** Individual sentences in the response.
6. **Words.** Individual words in the response.
7. **Characters.** Individual characters in the response.

Selection Operation. Given response units, the selector falls into four categories:

1. **Global** selects the entire response as a governed segment. For example, *The response must contain exactly three paragraphs* applies the constraint to the entire response.
2. **Traversal** selects every unit of a given kind. For example, *Each paragraph must contain at most three sentences* selects every paragraph.
3. **Position** selects units at specific locations. For example, *The first sentence must contain the keyword "summary"* targets a sentence by its position.
4. **Condition** selects units satisfying a predicate. For example, *Sentences containing "however" must contain at most 20 words* filters sentences by keyword occurrence.

Selection operations can be recursively composed to express nested scopes. For example, *The final sentence of every paragraph* combines **Traversal** over paragraphs with **Position** within each selected paragraph.

A.2 Target

The *Target* $\mathcal{T}_i$ specifies objects measured within the selected *Scope*. We organize 15 fine-grained subcategories into six major categories:

1. **Capacity** measures text length and natural-language units.
   (a) **Text Length.** The number of words or characters.
   (b) **Unit Count.** The number of natural-language units, such as paragraphs, sentences, or lines.
2. **Literals** track occurrences of specific strings or literal classes.
   (a) **Specific Strings.** Occurrences of exact characters, words, phrases, or sentences, such as the keyword "apple".
   (b) **Literal Classes.** Occurrences of characters or words belonging to a class, such as digits, uppercase letters, emoji, and punctuation marks.
3. **Structure** captures structured containers such as JSON or Markdown, their constituent members, and nesting depth.

   (a) **Containers.** Complete structured objects, such as JSON objects, Markdown lists, and CSV tables.
   (b) **Members.** Constituent elements within a structured object, such as JSON key-value pairs, table rows, and list items.
   (c) **Nesting Depth.** The hierarchical depth of a structured object.
4. **Checklist** assesses the inclusion of mandatory elements and valid values.
   (a) **Presence.** Inclusion of required components, such as JSON keys `type`, `time`, and `country`.
   (b) **Validity.** Fields or values satisfying predefined specifications, such as an integer `count` field or a decimal `price` field.
5. **Pattern** verifies conformity to boundary, skeleton, and surface templates.
   (a) **Boundary.** Objects delimited by defined separators, such as text enclosed in quotes or paragraphs separated by ***.
   (b) **Skeleton.** Strings matching a fixed-text skeleton with variable slots, such as `Question N: ... Answer: ...`.
   (c) **Surface.** Text spans matching a specific format, such as uppercase words or numbers with three decimal places.
6. **Relations** characterize multi-object dependencies.
   (a) **Ordering Relations.** Sequential arrangement of objects, such as names sorted alphabetically or events arranged chronologically.
   (b) **Chaining Relations.** Dependencies between adjacent elements, such as the last word of one sentence matching the first word of the next.
   (c) **Positional Relations.** Dependencies across designated positions, such as an acrostic formed by the first letters of consecutive lines.

A.3 RANGE

The *Range* $\mathcal{R}_i$ specifies the admissible values of *Target*. It comprises six categories: the first four are absolute, and the following two are relative:

1. **One-Sided Bounds.** Impose a minimum or maximum. For example, a response should contain at most 100 words.
2. **Prohibition.** Fix the measured value at zero. For example, a response must not contain punctuation marks.
3. **Equality.** Prescribe an exact value. For example, a response should contain exactly five sentences.
4. **Intervals.** Define a fixed closed numeric range. For example, a response should contain between 80 and 120 words.
5. **Comparison.** Impose an inequality relative to another response measurement. For example, each paragraph should contain more words than the preceding paragraph.
6. **Arithmetic Relations.** Enforce a precise equation relative to another response measurement. For example, the third paragraph should contain exactly twice as many words as the second paragraph.

This formulation also accommodates qualitative constraints. For example, constraints *The response should be a valid JSON object* or *The response should end with "happy birthday"* can be represented by defining a binary compliance *Target* with an **Equality** *Range* of one.

B LIST OF PROMPT TEMPLATES

This section lists all the prompt templates applied throughout this work, including the prompt for atomic constraint generation in Table 9, the prompt for constraint crossover in Table 10, the prompt for instruction quality assessment in Table 11, the prompt for constraint checklist quality assessment

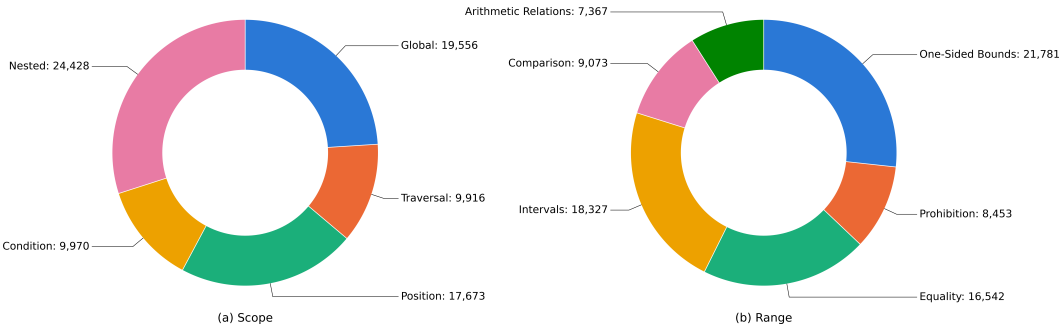


Figure 5: The *Scope* (a) and *Range* (b) distribution of constraints in SCOPEINSTRUCT.

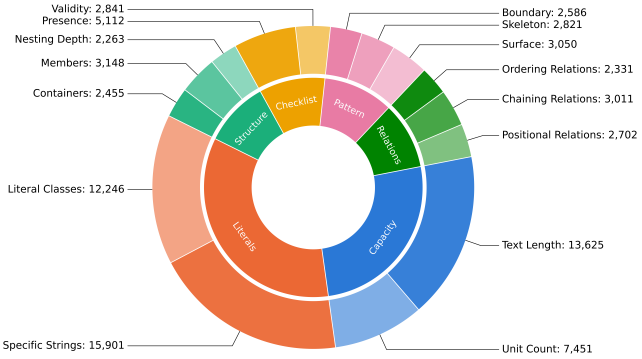


Figure 6: The *Target* distribution of constraints in SCOPEINSTRUCT.

in Table 12, the prompt for target extraction for each constraint in Table 13, the prompt for target extraction quality assessment in Table 14, the prompt for tool design in Table 15, and the prompt for tool-grounded judgment in Table 16.

C DATA STATISTICS OF SCOPEINSTRUCT

The constructed dataset, SCOPEINSTRUCT, contains 17,968 instructions, of which 16,968 are used for training and 1,000 for testing. The number of constraints per instruction ranges from 1 to 12, with an average of 4.83. Figure 8 shows the distribution of constraint counts. Grounded in our constraint schema, SCOPEINSTRUCT covers diverse constraint categories across *Scope*, *Target*, and *Range* dimensions, whose distributions are presented in Figures 5 and 6.

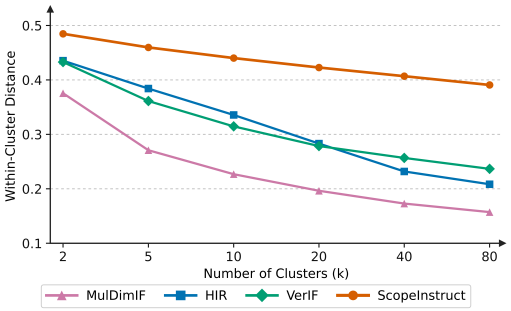


Figure 7: Mean squared distance from each constraint embedding to its assigned k-means centroid under varying numbers of clusters k.

As the results in Table 5 focus on n-gram variations, we further evaluate constraint diversity at the semantic level. Specifically, we extract constraint embeddings using BGE-M3[2] and apply k-means clustering to each dataset separately. We report the within-cluster distance, i.e., the mean squared distance from each constraint to its assigned centroid, where a lower value indicates that constraints are more templated and reducible to a limited number of prototypes. As shown in Figure 7, SCOPEINSTRUCT attains the highest within-cluster distance at every k and decays far more slowly. This suggests that constraints of baseline datasets collapse onto a limited set of recurring

[2]https://huggingface.co/BAAI/bge-m3

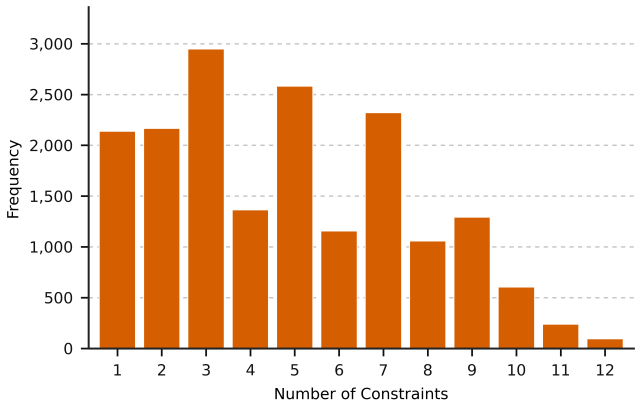


Figure 8: Distribution of the number of constraints per instruction in SCOPEINSTRUCT.

templates, while our factorized sampling over *Scope*, *Target*, and *Range* produces semantically dispersed combinations even under fine-grained partitioning.

D DETAILS OF REWARD CALCULATION

For each quantitative record $([l_{ij}, u_{ij}], a_{ij})$, the absolute deviation d_{ij} of the measured value a_{ij} from the permissible bound $[l_{ij}, u_{ij}]$ is calculated as:

$$d_{ij} = \max\{l_{ij} - a_{ij},\, a_{ij} - u_{ij},\, 0\} \tag{8}$$

Thus, $d_{ij} = 0$ when $a_{ij} \in [l_{ij}, u_{ij}]$; otherwise, it is the distance to the nearest range boundary. We first define $\bar{l}_{ij} = \max(l_{ij}, 0)$ and $\bar{u}_{ij} = \max(u_{ij}, 0)$, and calculate the characteristic range magnitude s_{ij} of $[l_{ij}, u_{ij}]$ as follows:

$$s_{ij} = \begin{cases} \max\left(1, \frac{\bar{l}_{ij}+\bar{u}_{ij}}{2}\right), & u_{ij} < +\infty \\ \max\left(1, \bar{l}_{ij}\right), & u_{ij} = +\infty \end{cases} \tag{9}$$

In addition, we employ a binary format reward that equals zero when a response is truncated at the maximum generation length or contains an unclosed `<think>` block, and one otherwise. When a reasoning block is present, only the final-answer text is used for constraint verification. The final training reward is obtained by multiplying this format reward by the hierarchical reward defined in Section 5.2:

$$R_{\text{final}}(x, y) = R_{\text{format}}(y) \cdot R(x, y) \tag{10}$$

For a fair comparison, we apply the same format reward to the RL-ILA and RL-CLA baselines.

E DETAILS OF EXPERIMENTS

E.1 EVALUATION METRICS

We evaluate precise instruction-following across four benchmarks: our test set, IFEval (Zhou et al., 2023), IFBench (Pyatkin et al., 2026), and IFHierBench (Mao & Chen, 2026). IFEval and IFBench employ four metrics: prompt- and instruction-level accuracy under both strict and loose evaluation conditions (P/I-S/L). Prompt-level accuracy requires all constraints to be satisfied, whereas instruction-level accuracy measures the proportion of satisfied constraints. Loose evaluation applies predefined normalizations to the response before verification. IFHierBench utilizes only prompt- and instruction-level strict accuracy (P/I-S). For our test set, following ImpRIF (Yang et al., 2026), we report CSR (Constraint Success Rate) and ISR (Instruction Success Rate). Given N instances, where the i-th instance contains an instruction x_i with constraints C_i and a corresponding response y_i, these metrics are computed as:

$$\text{CSR} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{|C_i|}\sum_{c\in C_i} I(x_i, y_i, c) \tag{11}$$

Table 6: CSR (%) across five *Scope* categories on our test set. The highest result for each backbone model is **bolded**, and the overall highest is underlined.

Model	Global	Traversal	Position	Condition	Nested
Frontier Models					
Seed-2.0-Pro	75.1	59.8	70.8	63.5	64.9
Gemini-2.5-Pro	71.5	56.6	60.8	61.0	56.9
DeepSeek-V3.2	61.5	45.7	45.1	44.2	43.1
GPT-4.1	62.5	49.2	49.0	46.3	45.6
QwQ-32B	58.3	40.4	45.5	43.3	39.9
Our Models					
Qwen3-4B	52.0	35.1	35.9	36.2	31.3
+ DPO	53.4	39.9	39.8	38.1	34.5
+ RL-ILA	56.9	45.9	43.4	46.2	42.9
+ RL-CLA	63.1	48.3	48.9	49.8	48.7
+ ScopeIF (Ours)	**65.6**	**53.8**	**53.2**	**59.0**	**56.3**
Qwen3-8B	53.7	40.8	39.5	39.0	32.7
+ DPO	54.5	41.5	40.3	41.3	35.9
+ RL-ILA	61.3	48.0	50.5	51.3	49.1
+ RL-CLA	66.4	56.1	57.2	56.0	56.0
+ ScopeIF (Ours)	**67.2**	**57.1**	**61.0**	**60.1**	**61.2**

$$\text{ISR} = \frac{1}{N}\sum_{i=1}^{N}\prod_{c\in C_i} I(x_i, y_i, c) \tag{12}$$

where $I(x_i, y_i, c)$ is an indicator function denoting whether response y_i satisfies constraint c. Finally, for general benchmarks MATH-500 (Lightman et al., 2024), GPQA (Rein et al., 2023), and MMLU-Pro (Wang et al., 2024), we report standard answer accuracy.

E.2 Implementation Details

For DPO training, we use the Llama-Factory (Zheng et al., 2024) framework, sampling 5 responses per instruction with a temperature of 1.0 and a top-p value of 0.9. The responses with the highest and lowest rewards are used to construct preference pairs. We use the Zero Redundancy Optimizer (ZeRO) (Rajbhandari et al., 2020) stage 3 from the DeepSpeed library (Rasley et al., 2020) and the AdamW (Kingma & Ba, 2014; Loshchilov & Hutter, 2017) optimizer with a weight decay of 0.1. The maximum sequence length is set to 8192 tokens. The peak learning rate is set to 1e-6 with a 10% warmup ratio and a cosine scheduler. The batch size is 128. Training utilizes a sigmoid loss function with a beta value of 0.1 and spans 1 epoch. Following previous works (Hou et al., 2024), an additional SFT loss is added to the chosen response with a weight of 0.1. The reward calculation and DPO training are both conducted on 8 H800 GPUs.

For GRPO training, we set the learning rate to 2e-6, with the maximum sequence length 4096 for prompts and 8192 for responses. The train-batch size and PPO mini-batch size are each set to 32. For rollout response generation, we employ the vllm (Kwon et al., 2023) framework, utilizing 50% of the available GPU memory, with a temperature and top-p value of 1.0. To encourage stable training, we also incorporate KL divergence regularization with a coefficient of 0.001, using the low-variance KL implementation. We save checkpoints every 30 steps during training. Following TULU 3 (Lambert et al., 2024) and VerIF (Peng et al., 2025a), we use IFEval (Zhou et al., 2023) as the validation set to select the best checkpoint. All experiments are conducted on 32 H800 GPUs, with 8 GPUs dedicated to model training and the remaining 24 GPUs allocated for API deployment of the judge model. The training process runs for one epoch, taking approximately 50 hours.

Table 7: CSR (%) across six *Target* categories on our test set.

Model	Capacity	Literals	Structure	Checklist	Pattern	Relations
Frontier Models						
Seed-2.0-Pro	61.4	69.5	79.1	74.2	74.6	58.2
Gemini-2.5-Pro	55.5	64.6	73.8	68.3	69.8	51.5
DeepSeek-V3.2	43.8	52.2	63.4	52.6	57.2	35.2
GPT-4.1	45.6	51.2	69.6	55.7	61.8	37.3
QwQ-32B	39.1	47.6	64.3	52.7	53.7	36.6
Our Models						
Qwen3-4B	35.4	41.9	47.1	44.0	46.9	26.8
+ DPO	37.6	44.2	53.7	49.0	47.6	28.6
+ RL-ILA	42.3	49.3	58.1	55.5	53.4	38.1
+ RL-CLA	44.2	55.8	68.8	58.5	62.8	39.7
+ ScopeIF (Ours)	**47.8**	**59.9**	**73.1**	**67.3**	**67.3**	**50.2**
Qwen3-8B	36.5	44.6	52.5	47.6	48.5	29.7
+ DPO	38.3	44.5	57.8	48.4	51.5	30.1
+ RL-ILA	44.4	54.4	65.3	60.5	61.8	43.6
+ RL-CLA	49.4	60.9	72.1	67.8	69.7	47.1
+ ScopeIF (Ours)	**53.7**	**65.1**	**72.9**	**69.4**	**70.8**	**50.4**

Table 8: CSR (%) across six *Range* categories on our test set.

Model	Absolute Range				Relative Range	
	One-Sided Bounds	Prohibition	Equality	Intervals	Comparison	Arithmetic Relations
Frontier Models						
Seed-2.0-Pro	80.3	82.0	61.0	61.5	68.9	40.2
Gemini-2.5-Pro	76.8	80.9	50.2	56.1	70.6	32.2
DeepSeek-V3.2	68.1	74.2	34.5	41.8	47.9	15.0
GPT-4.1	70.8	76.3	37.0	41.2	57.0	16.3
QwQ-32B	62.4	70.4	33.8	37.9	49.0	22.7
Our Models						
Qwen3-4B	56.7	65.4	26.8	27.9	38.7	12.3
+ DPO	60.4	68.8	30.1	28.4	42.2	12.3
+ RL-ILA	63.9	72.1	34.8	38.0	51.0	22.7
+ RL-CLA	67.6	72.8	41.8	44.8	56.1	29.1
+ ScopeIF (Ours)	**72.9**	**78.0**	**47.2**	**50.6**	**65.5**	**30.7**
Qwen3-8B	61.0	66.9	28.2	30.0	43.0	12.0
+ DPO	61.5	68.3	30.4	30.9	47.0	16.9
+ RL-ILA	67.1	75.3	41.3	45.2	53.6	30.4
+ RL-CLA	72.6	76.1	49.2	51.1	62.7	**37.4**
+ ScopeIF (Ours)	**75.2**	**79.2**	**52.8**	**56.5**	**63.8**	33.4

E.3 Detailed Results of Each Constraint Category

Tables 6, 7 and 8 present the CSR of each *Scope*, *Target* and *Range* category on our test set, respectively. For *Scope*, LLMs generally perform best on Global constraints, while their performance drops markedly on single-layer scope selectors (i.e., Traversal, Position, and Condition), and reaches its lowest point on recursively composed Nested scopes, demonstrating that scope complexity poses a growing challenge to LLMs. For *Target*, Capacity and Relations consistently emerge as the most challenging categories, suggesting that LLMs still struggle with precise control over text length and unit counts, as well as coordinating dependencies among multiple objects. For *Range*, One-Sided Bounds and Prohibition are comparatively easier, whereas Equality and Intervals pose greater challenges because they demand more precise numerical control. Notably, Arithmetic Relations prove particularly difficult, as they require models to maintain exact quantitative relationships among multiple targets, exposing a persistent weakness in jointly tracking and controlling coupled quantities across different response segments.

E.4 Details of Constraint Verification Methods

For pure LLM judges, we modify the judge prompt in Table 16 to exclude auxiliary verification results, requiring the judge model to directly output the final judgment alongside the detailed statistical information for each counting object. In instances where there is a discrepancy between the judgment result and the statistical information, we adopt the statistical information as the definitive result. For the rule-based verifier, we adjust the tool design prompt in Table 15, requiring the generated tool to take the entire response as input and yield a binary judgment on constraint satisfaction. To calculate pairwise agreement, we assign *win*, *tie*, or *lose* labels to a pair of responses for a given instruction based on the number of satisfied constraints evaluated by either humans or the judge model, and compute their agreement accordingly.

F Limitations

The limitations of our work are summarized as follows:

Potential Bias in Synthetic Data Construction. The construction of ScopeInstruct primarily relies on Seed-2.0-Pro for constraint generation and instruction synthesis, which may introduce model-specific preferences. Although we employ a different LLM, GPT-OSS-120B, for quality inspection to mitigate this issue, incorporating multiple data-generation models and broader human auditing may further improve data robustness and diversity. We consider this an important direction for future work.

Cost of Tool-Grounded Verification. ScopeIF employs judge models to dynamically design response-specific tools for constraint verification, introducing additional computational overhead during reward calculation. We consider this overhead a necessary trade-off for reliably isolating relevant response segments and measuring complex targets that elude both a fixed inventory of rule-based verifiers and pure LLM judges. Importantly, this additional cost is incurred only during the training stage and does not affect the inference efficiency of the resulting policy models. Nonetheless, further reducing this overhead through lightweight judge models, batched verification, and reusable verification tools for recurring constraint patterns remains an important future work.

Table 9: The prompt template for atomic constraint generation.

I need to construct diverse and challenging objective constraints. In this task, each objective constraint is represented by three dimensions: *Scope*, *Target*, and *Range*. They dictate the permissible bounds of specific measurable objects within designated response segments. The taxonomy is provided as follows:
{constraint schema}

I will provide you with a user instruction and a specific constraint category that contains the combination of these three dimensions. Your task is to add one or more suitable objective constraints to the user instruction that conform to the specified constraint category to construct new instructions. You must provide five distinct construction results. I will also provide previously generated instructions for the same category. You must ensure that the newly added constraints in the five results differ as much as possible from those in the previous results and remain diverse. The specific constraint category will be provided in the following format:

```
{
  "Scope": ... (one of "Global", "Traversal", "Position", and "Condition"),
  "Nested Scope": ... (true or false),
  "Target": ... (one of "Capacity", "Literals", "Structure", "Checklist", "Pattern", and
"Relations"),
  "Range": ... (one of "One-Sided Bounds", "Prohibition", "Equality", "Intervals",
"Comparison", and "Arithmetic Relations")
}
```

When completing this task, you must follow the principles below:
1. To integrate the original instruction and new constraints more naturally and coherently, you can appropriately revise the instruction. Objective constraints that are already present in the original instruction should preferably be removed or replaced. Moreover, you should focus on adding objective constraints and avoid subjective constraints such as topic or style. The newly constructed instruction must use the same language as the original instruction.
2. The newly added constraints must be natural, clear, understandable, reasonable, practical, and suitable for the new user instruction. They must not be contrived or impossible to fulfill. The objective constraints must not conflict with one another or with any other part of the instruction.
3. The new user instruction must contain complete information and must not omit any necessary content. You must ensure that a model can understand the meaning and complete information of the new instruction and fulfill its task when the original instruction is not visible.
4. The newly added constraints must be sufficiently challenging for LLMs. Thus, you should carefully design the *Range* according to the characteristics of the user instruction. As a general criterion, without the added constraint, LLMs' output should have a high probability of violating it, and satisfying the added constraint should require LLMs to adjust their response deliberately.
5. The newly added constraints in all five results must conform to the given constraint category.
6. The newly added constraints in all five results must differ from those in the previously generated instruction while also remaining diverse from each other. The response unit, implementations of *Scope* selector, secondary *Target* categories, *Range* values, and other unspecified details should be as diverse as possible. The final constraints should also be expressed in diverse ways. I will provide the distributions of response units and secondary *Target* categories for constraints from previously constructed instructions. You should prioritize constraints that have received less coverage when constructing the constraints for the current round.
7. The newly added constraints should use natural and varied expressions rather than being mechanically phrased as count-based statements. A qualitative constraint (e.g., valid JSON format or a specific text ending) is a special case of an Equality *Range* of one.
8. The newly added constraints should be concise and clear. Avoid appending commonsense explanations, definitions, or examples.
9. You are not limited to adding only one constraint and are encouraged to add multiple constraints from the given category to construct a more complex constraint. If multiple constraints are added, they must use closely related implementations either as parallel instances of the same atomic constraint or as constraints with a clear logical progression. They must not be unrelated constraints with substantially different forms. Their category must be the same.

You must construct five high-quality new instructions and provide both the new instructions and the newly added constraints. Strictly follow the output format below:

```python
**[The Start of Analysis]**
... (Provide a detailed analysis and plan explaining how to construct new instructions by adding diverse, high-quality, challenging objective constraints that differ from previously generated constraints and conform to the specified constraint category)
**[The End of Analysis]**

**[The Start of Construction Results]**
[
  {
    "Original Instruction": ... (string; directly copy the original instruction without any
modifications),
    "New Instruction": ... (string; the constructed new instruction),
    "Newly Added Constraints": ... (string; all constraints added to the original
instruction. Directly extract segments from the new instruction without any modifications.
The extraction must be complete and cover all added constraints),
    "Scope": ... (string; use the format "Scope Selector(Response Unit)". The Scope
Selector must be one of "Global", "Traversal", "Position", and "Condition". The Response
Unit must be one of "Entire Response", "Structural Block", "Structural Element", "Paragraph",
"Sentence", "Word", and "Character". Connect nested Scope with " -> "),
    "Secondary Target Category": ... (string; one of Secondary Target Category)
  },
  ... (a list of dictionaries containing exactly five items)
]
**[The End of Construction Results]**
```

The user instruction, the specific constraint category, and the previously constructed instruction are provided below:

[The Start of User Instruction]
{instruction}
[The End of User Instruction]

[The Start of Constraint Category]
{category}
[The End of Constraint Category]

[The Start of Previous Constructed Instructions]
{existing_instructions}
[The End of Previous Constructed Instructions]

[The Start of Previous Response Unit Distribution]
{existing_response_unit_distribution}
[The End of Previous Response Unit Distribution]

[The Start of Previous Secondary Target Category Distribution]
{existing_secondary_target_category_distribution}
[The End of Previous Secondary Target Category Distribution]

Table 10: The prompt template for constraint crossover.

I need to construct diverse and challenging objective constraints. In this task, each objective constraint is represented by three dimensions: *Scope*, *Target*, and *Range*. They dictate the permissible bounds of specific measurable objects within designated response segments. The taxonomy is provided as follows:
{constraint schema}

I will provide you with a user instruction and several objective constraints. Your task is to construct five more complex new instructions by adding suitable objective constraints to the user instruction in distinct ways. For each new instruction, you must add at least {constraint_number} constraints. The same constraint may be appropriately modified and added multiple times. The constructed instructions should be diverse and differ as much as possible. Each given constraint consists of an ID, its content, and its classification under the taxonomy described above. The format is as follows:

```
{
   "Constraint ID": ... (integer),
   "Constraint Content": ... (string),
   "Constraint Classification": {
      "Scope": ... (string),
      "Primary Target": ... (one of "Capacity", "Literals", "Structure", "Checklist",
"Pattern", and "Relations"),
      "Secondary Target": ... (string),
      "Range": ... (one of "One-Sided Bounds", "Prohibition", "Equality", "Intervals",
"Comparison", and "Arithmetic Relations")
   }
}
```

When completing this task, you must follow the principles below:
1. To integrate the original instruction and new constraints more naturally and coherently, you can appropriately revise the instruction. Objective constraints that are already present in the original instruction should preferably be removed or replaced. Every newly added constraint must be obtained by adjusting or modifying one of the given constraints. You are prohibited from inventing new constraints unrelated to the given constraints. Moreover, you should focus on adding objective constraints and avoid subjective constraints such as topic or style.
2. The newly constructed instructions must use the same language as the original user instruction. You must not add constraints that are specific to other languages and different from the newly constructed instructions.
3. The new user instruction must contain complete information and must not omit any necessary content. You must ensure that a model can understand the meaning and complete information of the new instruction and fulfill its task when the original instruction is not visible.
4. The newly added constraints must be natural, clear, understandable, reasonable, practical, and suitable for the new user instruction. They must not be contrived or impossible to fulfill. The objective constraints must not conflict with one another or with any other part of the instruction.
5. The newly added constraints must be sufficiently challenging for LLMs. Thus, you should carefully design the *Range* according to the characteristics of the user instruction. As a general criterion, without the added constraint, LLMs' output should have a high probability of violating it, and satisfying the added constraint should require LLMs to adjust their response deliberately.
6. The newly added constraints in all five results must be as diverse as possible and should collectively cover a diverse selection of the given constraints. The final constraints should also be expressed in diverse ways.
7. The newly added constraints should be concise and clear. Avoid appending commonsense explanations, definitions, or examples.
8. To reiterate, the same given constraint is not limited to being added only once. You are encouraged to modify and add the same constraint multiple times to construct more complex constraint forms.

You must generate five high-quality new user instructions and provide each new instruction together with all objective constraints it contains. Strictly follow the output format below:

```
**[The Start of Analysis]**
... (Provide a detailed analysis and plan explaining how to construct new instructions by adding diverse, high-quality, and challenging new constraints)
**[The End of Analysis]**
```

[The Start of Construction Results]

```
[
  {
    "Instruction ID":  ...  (integer from 1 to 5),
    "New User Instruction":  ...  (a new user instruction constructed by adding suitable
constraints to the original instruction),
    "All Added Constraints in the New User Instruction":  [
      {
        "Constraint Content":  ...  (string; completely extract one newly added constraint
contained in the new instruction),
        "Source Constraint ID":  ...  (integer)
      },
      ...  (extract all newly added constraints in the new user instruction without any
omissions.  If the same given constraint is used multiple times, output multiple items with
the same Source Constraint ID rather than merging them into one item)
    ]
  },
  ...  (a list containing exactly five items, each of which contains a new user instruction
and its newly added constraints)
]
```

[The End of Construction Results]

The user instruction and given constraints are provided below:
[The Start of User Instruction]
{instruction}
[The End of User Instruction]

[The Start of Given Constraints]
{given_constraints}
[The End of Given Constraints]

Table 11: The prompt template for user instruction quality assessment.

You are a professional artificial intelligence expert proficient in machine learning and natural language processing, with expertise in assessing the quality of user instructions. I will provide you with a user instruction. You must carefully analyze its logic and assess its quality. When assessing the quality of a user instruction, a high-quality instruction must satisfy all of the following conditions. Violation of any one condition means that the instruction should be considered low-quality. You must carefully analyze the instruction word by word to determine whether it satisfies each of the following conditions:

1. The intent is completely explicit.
2. The semantics are completely clear, and the language is coherent, fluent, and easy to understand, without ambiguity, incomprehensible statements, or grammatical errors.
3. It contains no erroneous content or incorrect information.
4. It does not exceed the capabilities of an offline text-only model; an offline text-only model must be able to complete the task. Being offline means that the model cannot access niche knowledge, such as obscure books, files, materials, or other information, nor can it access specialized knowledge, such as professional engineering knowledge or specialized books.
5. It contains no impossible constraints. Note that unconventional, highly difficult constraints are permitted, provided that they are not impossible to fulfill.
6. It contains no erroneous or disorganized logic.
7. Its internal logic is consistent, with no contradictory logic or constraints. No parts of the instruction conflict with or contradict one another, and all constraints can be fulfilled simultaneously.
8. The instruction and its accompanying materials are complete, with no missing content or data.
9. The user instruction provides all necessary data and information.
10. The user instruction must not contain any problem that requires multi-turn interaction to resolve. In other words, the AI assistant must be able to respond to the instruction completely within a single turn.
11. The user instruction is concise without redundant explanations.

You are prohibited from answering the user instruction. You only need to determine whether it is high-quality or low-quality. Strictly follow the format below when providing your analysis and conclusion:

```
**Analysis:** ... (Provide a comprehensive, detailed, and meticulous analysis)
**Final Conclusion:** ... (The final conclusion must be either "[[High Quality]]" or "[[Low Quality]]")
```

Here is the user instruction:
[The Start of User Instruction]
{instruction}
[The End of User Instruction]

Table 12: The prompt template for constraint checklist quality assessment.

You are a professional artificial intelligence expert proficient in machine learning and natural language processing, with expertise in assessing the quality of user instructions. I will provide you with a user instruction containing objective constraints, together with the extraction results for those constraints. You must carefully analyze and thoroughly examine whether the extraction results are correct. Objective constraints dictate the permissible bounds of specific measurable objects within designated response segments. The taxonomy is provided as follows: {constraint schema}

A high-quality extraction result must satisfy all of the following conditions. Violation of any one condition means that the extraction result should be considered low-quality. You must carefully analyze the extraction results word by word to determine whether they satisfy each of the following conditions:

1. All objective constraints in the user instruction are completely extracted, without containing any other constraints, and none of them are omitted.
2. All objective constraints in the user instruction are accurately extracted, and the meaning of each extracted constraint is completely consistent with the instruction.
3. Each objective constraint is extracted exactly once, with no duplicate extractions.
4. Each extracted objective constraint is expressed clearly, without any ambiguity.
5. There is no restriction on extraction granularity. Multiple constraints may either be combined into a single extracted item or divided into multiple extracted items.

You are prohibited from answering the user instruction. You only need to determine whether the extraction result is high-quality or low-quality. Strictly follow the format below when providing your analysis and conclusion:

```
**Analysis:** ... (Provide a comprehensive, detailed, and meticulous analysis)
**Final Conclusion:** ... (The final conclusion must be either "[[High Quality]]" or "[[Low Quality]]")
```

The user instruction and its corresponding objective constraint extraction results are provided below:
[The Start of User Instruction]
{prompt}
[The End of User Instruction]

[The Start of Extraction Results]
{constraint_checklist}
[The End of Extraction Results]

Table 13: The prompt template for target extraction for each constraint.

You are an expert skilled at extracting important information from instructions. I will provide you with a user instruction and a given constraint contained in that instruction. My goal is to determine whether some responses satisfy the given constraint. Your task is to completely extract all counting objects of the given constraint. You must follow the principles below when completing this task:

1. Counting objects can take two forms. Every counting object must be expressed using the first form whenever possible. The second form may be used only when the counting object cannot be expressed using the first form.
    a. If the subconstraint directly dictates the quantity of certain content, the counting object should be the quantity of that content, excluding the specific quantitative constraint.
    b. If the subconstraint cannot be expressed using the above form, the counting object should be expressed in a qualitative form: a question that can be answered with "yes" or "no" and indicates whether the corresponding subconstraint is satisfied.
2. The counting object must be divided comprehensively and without omission, and every part of the given constraint that needs to be verified must have a corresponding counting object. The counting object must correspond exactly to the content of the given constraint. Every subconstraint must have a corresponding counting object, and the counting object must not contain anything unrelated to the given constraint. In other words, every counting object must fall within the scope of the given constraint, and the counting objects must collectively constitute a necessary and sufficient condition for satisfying the given constraint. Furthermore, the counting object should be divided as atomically as possible, and complex constraints should be divided into multiple counting objects whenever possible.
3. You must distinguish whether each counting object is universal or specific. "Universal" means that the corresponding subconstraint applies to an unspecified number of multiple objects, whereas "specific" means that it applies to exactly one particular object. You must provide the type of every counting object in the final output. A specific counting object must correspond to the statistical result of exactly one particular object and must not correspond to multiple statistical results.
4. If the quantity in a constraint is expressed using the quantity of another reference object rather than a specific number, use the subject of the constraint as the counting object. The reference value must not be treated as a counting object.
5. The counting object must use the same language as the user instruction.

Use the following output format:

````
**[The Start of the Constraint]**
... (It must be identical to the constraint I provide, without any modification)
**[The End of the Constraint]**

**[The Start of Counting Object]**
```json
[
````
````

```
  {
    "Counting Object":  ...  (Describe the counting object in detail, using the original
wording of the constraint wherever possible),
    "Type":  ...  ("Universal" or "Specific")
  },
  ...
]
```
[The End of Counting Object]
````

The user instruction and the given constraint contained in it are provided below:
**[The Start of User Instruction]**
{instruction}
**[The End of User Instruction]**

**[The Start of the Constraint]**
{constraint}
**[The End of the Constraint]**

Table 14: The prompt template for target extraction quality assessment.

You are a professional artificial intelligence expert proficient in machine learning and natural language processing, with expertise in assessing the quality of user instructions. I will provide you with a user instruction, a given constraint contained in that instruction, and a list of counting objects for this constraint. Your task is to carefully analyze whether the object list is completely correct and satisfies all of the following conditions, and ultimately conclude with either "[[High Quality]]" or "[[Low Quality]]". High-quality counting object lists must satisfy all of the following conditions. Violation of any one condition means that the list should be considered low-quality. You must carefully analyze the object list word by word to determine whether it satisfies each of the following conditions:

1. The statistical result of every counting object must be representable either by a number or by "yes/no".
2. Every counting object must be correctly classified as either "Universal" or "Specific". "Universal" means that the corresponding subconstraint applies to an unspecified number of multiple objects, whereas "Specific" means that it applies to exactly one particular object. A specific counting object must not correspond to multiple statistical results. The type of every counting object must be provided.
3. If the quantity in a constraint is expressed using the quantity of another reference object rather than a specific number, use the subject of the constraint as the counting object. The reference value must not be treated as a counting object.
4. The counting objects must correspond exactly to the content of the given constraint. Every subconstraint of the given constraint must have a corresponding counting object, and the counting objects must not contain anything unrelated to that constraint. In other words, every counting object must fall within the scope of the given constraint, and the counting objects must collectively constitute a necessary and sufficient condition for satisfying the given constraint.

Strictly follow the format below when providing your analysis and conclusion:

```
**Analysis:** ... (Provide a comprehensive, detailed, and meticulous analysis)
**Final Conclusion:** ... (The final conclusion must be either "[[High Quality]]" or "[[Low Quality]]").
```

The user instruction, the corresponding constraint, and the counting object list are provided below:
**[The Start of User Instruction]**
{instruction}
**[The End of User Instruction]**

**[The Start of the Constraint]**
{constraint}
**[The End of the Constraint]**

**[The Start of Counting Object List]**
{target}
**[The End of Counting Object List]**

Table 15: The prompt template for tool design.

You are an expert skilled at analyzing the quality of AI assistant responses. I will provide you with a user instruction, a given constraint contained in that instruction, a response, and a list of counting objects for this constraint. My goal is to determine whether the response satisfies the given constraint. For this constraint, generate a Python auxiliary verification tool to help me make a more accurate judgment. You must follow the principles below:

1. Auxiliary verification means using a tool to automatically obtain important information from the response to assist a subsequent judge model to verify this constraint. It does not mean using code to directly produce a judgment about whether the constraint is satisfied.
```

2. The generated auxiliary verification tool must satisfy the following requirements:
   a. The auxiliary verification tool should be generated only for completely objective constraints. Subjective constraints must not be verified with code. However, not every objective constraint necessarily requires an auxiliary verification tool.
   b. The information computed by the auxiliary verification tool must be fully discriminative and deterministic. The computed information must differ significantly between responses that satisfy the constraint and responses that do not.
   c. The information computed by the auxiliary verification tool should cover the list of counting objects as comprehensively as possible without being limited to it. An object in the list may be omitted if it is subjective or difficult to compute accurately with code. You may also freely compute other information outside the list that you consider necessary.
3. The auxiliary verification tool must be a Python function satisfying the following requirements:
   a. You can freely design the function's input parameters, but their values must be response segments relevant to the constraint.
   b. The function must return a list. Each item in the list must be a Dict containing one category of important information obtained by the auxiliary verification tool. Each Dict must follow this format:

```
{
  "Description": ... (string; provide a detailed definition of the information obtained by the tool so that the judge model can understand and use it),
  "Computed Result": ... (any format; provide the computed information),
}
```

   c. All Python packages that need to be imported must be imported inside the function. The function must not use any Python package that requires an additional download.
4. To ensure a consistent standard when counting words or characters or when splitting sentences, directly use the following functions implemented by human experts unless the constraint explicitly specifies otherwise:
   a. `count_word(s: str) -> int`: Counts the words or Chinese characters in a string. Its input is the string to be counted, and it returns the result as an integer. Use the `len(s: str) -> int` function only when the constraint explicitly specifies a character count.
   b. `split_sentences(s: str) -> List`: Splits a string into sentences. Its input is the string, and it returns a list containing all sentences in the string.
5. If you choose to generate an auxiliary verification tool, you must also extract the corresponding value of each input parameter from the response according to the following principles:
   a. You should extract only continuous verbatim segments from the response. You are strictly prohibited from modifying, adding, or deleting any content.
   b. If the extracted value corresponding to a parameter is the complete response, output `"ALL"` instead of reproducing the entire response. If no corresponding value exists in the response, output `""`.

Use the following output format:

````
[The Start of Auxiliary Verification Tool]
```python
... (Provide a Python tool that can assist in verifying the constraint according to the specifications above. If you choose not to generate an auxiliary verification tool, output "None" here.)
```
[The End of Auxiliary Verification Tool]

[The Start of Input Parameters]
... (Provide the name, format, description, and value of every input parameter designed for the auxiliary verification tool in detail. List the parameters in the same order as they appear in the function and use the JSON format below. If you choose not to generate an auxiliary verification tool, output "None" here.)
```json
[
  {
    "Parameter Name": ... (the parameter name, which must match the name used in the auxiliary verification tool),
    "Format": ... (the parameter's data type),
    "Parameter Description": ... (the description and meaning of the parameter),
    "Parameter Value": ... (the value corresponding to the parameter in the response. It must be a complete verbatim segment from the response and must not be modified in any way. Its format must be consistent with the input parameter definition)
  },
  ...
]
```
[The End of Input Parameters]
````

The user instruction, the corresponding constraint, the response, and the counting object list are provided below:
[The Start of User Instruction]
{instruction}
[The End of User Instruction]

[The Start of the Constraint]
{constraint}
[The End of the Constraint]

[The Start of the Response]
{response}
[The End of the Response]

[The Start of Counting Object List]
{target}

[The End of Counting Object List]

Table 16: The prompt template for tool-grounded judgment.

You are an expert skilled at evaluating the quality of responses generated by artificial intelligence assistants. I will provide you with a user instruction, a constraint contained in that instruction, a response to the instruction, and a list of counting objects for this constraint. You must carefully examine whether the response satisfies the corresponding constraint in the user instruction and explain your reason. I will also provide some reference information (may be empty), which is automatically calculated by auxiliary verification tools for the corresponding constraint. You should use this information cautiously when making your final judgment and provide a counting result for every counting object. Note that auxiliary verification tools have limitations, so the reference information it provides is not guaranteed to be completely correct and may contain errors. If you are confident that the reference information contains an obvious error, you may disregard it. You must follow the principles below:

1. Your judgment should be as strict as possible. You should conclude "[[The response satisfies the constraint]]" only when the response completely satisfies every part of the corresponding constraint. If the response contains any omission or error in fulfilling the constraint, you must conclude "[[The response does not satisfy the constraint]]".
2. Your judgment of the corresponding constraint must remain independent. When evaluating the current constraint, you do not need to consider whether other parts of the user instruction are satisfied.
3. After providing your judgment, combine the reference information with your analysis and provide a counting result for every item in the counting object list. Follow the principles below when providing the counting results:
    a. Pay attention to whether each counting object is "Universal" or "Specific". For a specific counting object, use the original wording of that counting object. For a universal counting object, do not directly use its original wording. Instead, enumerate a separate counting result for every corresponding item that appears in the response.
    b. Provide counting results only for the items included in the counting object list.
4. For some constraints, the permissible bounds are not specified using a concrete number but are instead expressed through a relative quantitative relationship with a reference value, such as an inequality or arithmetic relationship. In such cases, you must first calculate the reference value and then substitute it into the permissible bounds.
5. You must ensure that your conclusion and counting results are logically consistent. If your judgment is "[[The response satisfies the constraint]]," the actual value of every counting result must fall within its permissible bounds range. If your judgment is "[[The response does not satisfy the constraint]]," the actual value of at least one counting result must fall outside its permissible bounds range.

You must strictly follow the format below when providing your analysis and judgment of the constraint:

````
**Constraint:** ... (Provide the original corresponding constraint verbatim, without making any modification)
**Analysis:** ... (Provide a detailed and meticulous analysis based on the specific content of the response, explaining whether the response satisfies the constraint)
**Conclusion:** ... (This must be either "[[The response satisfies the constraint]]" or "[[The response does not satisfy the constraint]]")
**Counting Results:** ```json
[
  {
    "Counting Object": ... (A string that describes the counting object in detail),
    "Permissible Bounds": ... (A two-element list specifying a closed interval in the form [a, b]. The two endpoints may be equal when only one value is valid. For qualitative constraints, the permissible bounds may be set to [0, 0] or [1, 1]),
    "Actual Value": ... (An integer specifying the actual value of this counting object in the response),
  },
  ...
]
```
````

The user instruction, the corresponding constraint, the auxiliary verification results, the response, and the counting object list are provided below:

[The Start of User Instruction]
{instruction}
[The End of User Instruction]

[The Start of the Constraint]
{constraint}
[The End of the Constraint]

[The Start of Auxiliary Verification Results]
{verification_results}
[The End of Auxiliary Verification Results]

[The Start of the Response]
{response}
[The End of the Response]

[The Start of Counting Object List]
{target}
[The End of Counting Object List]
````