IFHIERBENCH: HIERARCHICAL INSTRUCTION FOLLOWING FOR LARGE LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Instruction-following ability is critical for deploying large language models in real-world applications, where downstream components depend on the output satisfying specific constraints. Modern deployments increasingly handle the full task in a single LLM call, with one prompt specifying a layered output whose overall artifact, structural sections, and nested fields must each satisfy concrete constraints. Existing instruction-following benchmarks treat the constraint set as a flat list applied uniformly to the response, so they cannot scope a check to a particular section of the output. We introduce IFHierBench, a hierarchical instruction-following benchmark of 600 prompts stratified across four constraint-tree depths and 35 distinct constraints, each prompt paired with a deterministic checker that verifies satisfaction at every scope. Evaluating seven leading proprietary and open-weight models, we find that even the strongest model only marginally exceeds 50% prompt-level accuracy and that accuracy degrades sharply as constraint depth grows. Reliably following nested constraints remains a substantial gap for current LLMs, motivating future training methods that consider constraint adherence at finer granularity to achieve better instruction-following ability.

1 INTRODUCTION

Large language models (LLMs) can produce fluent text. Yet semantic correctness alone is not sufficient: they must also follow user-specified instructions. These instructions are often output constraints (Liang et al., 2024), and an empirical study of real-world developer prompts finds that over 80% of them are objectively verifiable: required fields, format, length, and required or forbidden keywords (Liu et al., 2024). Such constraints admit deterministic checking and so define a measurable instruction-following task. How users specify these constraints has changed with what models and agent frameworks can support. When a task exceeded what a single call could do, users worked *bottom-up*: decompose the task into small steps (Khot et al., 2022), prompt for each fragment, and let application code assemble the fragments into the final artifact. Each call's constraints governed that call's entire output, so treating a constraint as a property of the whole response was accurate. As model capability grew and agent frameworks matured (Yang et al., 2024), the workflow inverted. Users now hand the model the whole task and specify the artifact *top-down*: the overall format, the sections inside it, and the fields nested inside those. The model must realize this nested specification in a single pass, even inside agent pipelines, where one call now returns a complete structured artifact rather than a fragment. Constraints did not just grow in number, they moved inside the output: each one now governs a region of the response, and the decomposition that pipeline code once performed has become the model's job. Whether current LLMs can carry this burden is the question we study.

Figure 1 shows what such a top-down specification looks like. The prompt asks for a financial analysis report as a Markdown document with three H2 sections, requires two H3 subsections inside the Executive Summary with a bullet count and a keyword bound to Highlights, and requires a JSON code block inside Methods with key and value constraints reaching three levels deep. Most of the instruction-tuning data in common use today still mirrors the old workflow: fragment-sized tasks with constraints that govern the entire output. Here the whole specification lands in one prompt, and the model must satisfy every constraint in its scope. The response in the figure does not: the bullets and the keyword land under Outlook instead of Highlights, and the JSON block lands under Risk Assessment instead of Methods, so the key and value constraints that depend on it can no longer be

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.

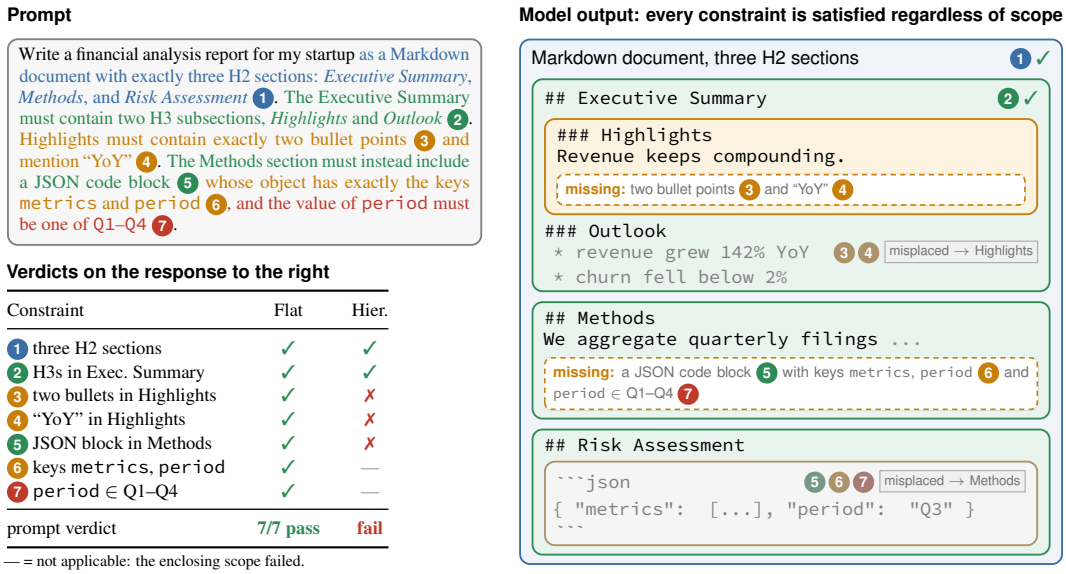


Figure 1: An example of hierarchical instruction following. Colored spans in the prompt (left) are constraints. The response (right) satisfies the content of every constraint regardless of scope: a flat checker incorrectly scores it 7/7, while the hierarchical evaluation of IFHierBench fails it as expected.

satisfied. A miss at an outer scope defeats every constraint built on top of it. In the figure, misplaced content is greyed and a dashed slot marks where each piece belongs. Real prompts wrap more layers.

Existing benchmarks for testing model instruction following treat the constraints in a prompt as a single flat list, with every check applied uniformly to the whole response (Zhou et al., 2023; Wen et al., 2024; Pyatkin et al., 2026). Among them, IFEval and IFBench impose independent checks on the whole response, and ComplexBench adds logical relations such as conjunction, chaining, and selection, yet none scope a check to a nested region of the output. On this single-list formulation they are already near saturation, with leading models including ones under 10B parameters scoring above 80% on IFEval (Yang et al., 2025; Guo et al., 2025; Zeng et al., 2025), while their behavior on prompts with the kind of nested scoping just described remains untested. Treating such a hierarchy as a flat list does more than coarsen the measurement, it distorts it: a flat checker verifies that each requirement is satisfied somewhere, not that it is satisfied where the prompt scoped it. This is exactly what happens in Figure 1: the flat verdict on the response is a perfect 7 of 7, while the hierarchical verdict is a fail spread across two depths. Flat evaluation certifies exactly the responses that break downstream consumers, and prior benchmarks cannot express the checks that would catch them.

Table 1: Comparison with prior benchmarks.

Benchmark	Relation	Verifier	Size
IFEval	Independent	Code	541
IFBench	Independent	Code	300
ComplexBench	Logical	Rule+LLM	1,150
IFHierBench	**Hierarchical**	**Code**	**600**

In this work, we introduce IFHierBench, a hierarchical instruction-following benchmark of 600 prompts spanning constraint depths 0 through 3, with 150 prompts at each depth. Table 1 compares IFHierBench with prior instruction-following benchmarks: it is the only one that scopes checks to nested output regions while verifying every check with deterministic code. The benchmark covers 35 distinct constraints. Of these, 10 structure-level constraints over JSON, Markdown, list, and labeled-section scopes are derived from an empirical analysis of output requirements across 1,232 real LLM application prompts from GitHub, while the remaining 25 content-level constraints are drawn from IFEval (Zhou et al., 2023) and ComplexBench (Wen et al., 2024). A data-synthesis pipeline turns each IFEval seed task into a constraint tree through depth-first sampling gated by

a sibling-conflict matrix, then renders the tree into a single natural-language prompt by merging compatible sibling constraints into joint sentences. We pair each constraint type with a deterministic Python checker, and the pipeline composes these per-type checkers along the constraint tree to emit a hierarchical checker for every prompt, so that satisfaction at every scope is verified without any model-based judging. Evaluating seven leading models, we find that even the strongest model only marginally exceeds 50% prompt-level accuracy, and accuracy degrades sharply with depth.

Our contributions are listed as follows:

- A new hierarchical instruction-following benchmark of 600 prompts stratified across four constraint depths, with 35 distinct constraints (10 structure-level and 25 content-level), where the structure-level set is derived from an empirical analysis of 1,232 real LLM application prompts from GitHub.
- A data synthesis pipeline that turns real-world tasks into hierarchical constraint trees and renders them as natural-language prompts paired with a deterministic hierarchical checker.
- Experiments on seven leading models showing that even the strongest model only marginally exceeds 50% prompt-level accuracy, and that accuracy degrades sharply with depth.

2 Related Work

2.1 Controlled Text Generation

A large body of work has pushed language models to follow user-specified constraints more reliably, through both *decoding-time* biasing and *training-time* fine-tuning. Decoding-time methods either force required tokens or phrases to appear by constraining the beam-search frontier (Hokamp & Liu, 2017; Post & Vilar, 2018), or use auxiliary attribute discriminators and future-aware predictors to bias the next-token distribution at each step (Dathathri et al., 2019; Khanov et al., 2024; Shin et al., 2025; Yang & Klein, 2021; Lu et al., 2021). Training-time approaches bake constraint adherence into the model itself, by conditioning generation on control codes prepended to the input (Keskar et al., 2019; Krause et al., 2021) or by aligning the model to human preferences through instruction tuning (Ouyang et al., 2022). These methods all operate on *flat, single-scope* constraints that act on the response as a whole. Whether they transfer to the *hierarchical* constraints of real prompts, where a format choice at one scope licenses restrictions on the scopes nested inside it, remains open. IFHierBench supplies training and evaluation data in this hierarchical regime.

2.2 Instruction-Following Benchmarks

Instruction-following benchmarks evaluate how reliably LLMs satisfy user-specified output constraints. One line enumerates as many distinct constraint types as possible to broadly test instruction-following ability: IFEval (Zhou et al., 2023) fixes 25 verifier templates, FollowBench (Jiang et al., 2024) stacks five constraint families along a difficulty ladder, IFBench (Pyatkin et al., 2026) adds previously unseen types to test generalization, and CFBench (Zhang et al., 2025) pairs 200+ scenarios with checklist scoring and constraint priorities. A second line studies how multiple constraints interact inside one prompt: ComplexBench (Wen et al., 2024) attaches explicit logical operators such as conjunction, chain, and selection between constraints. A third line extends instruction-following to multi-turn dialogues (Bai et al., 2024; Kwan et al., 2024; Li et al., 2025). None of these benchmarks treats the output of a single turn as a nested structure with constraints scoped to its sub-regions, and most emphasize content-level requirements rather than structure-level correctness. IFHierBench fills both gaps by providing prompts whose constraints form a tree over the output's structural geometry, evaluating format-level correctness at outer scopes and content-level constraints within them.

3 Structure-level Constraint Taxonomy

3.1 Constraint Collection

Our corpus is built on PromptSet (Pister et al., 2024), a collection of prompts extracted from LLM-based applications (LLMapps) in open-source GitHub projects. These projects span a wide range

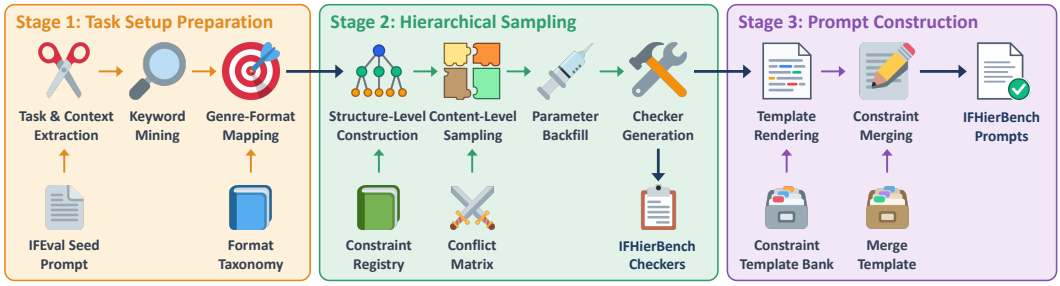


Figure 2: Overview of the IFHierBench construction pipeline.

of use cases and adoption levels, from personal demos to widely deployed systems, so prompt quality varies substantially. To reflect mature practice, we focus on widely used prompt templates (predefined structures that combine static text with dynamic placeholders to create adaptable prompts for LLMapps) in these apps (Schulhoff et al., 2024; Zhao et al., 2025). We then apply the cleaning pipeline of Mao et al. (2025) that filters by repository popularity and prompt length, and segment each remaining prompt into different components. We retain only the prompts that contain at least one explicit output constraint. This yields 1,232 prompt templates from 2,888 prompts in 1,525 repositories.

3.2 Annotation and Taxonomy

We use GPT-5.4 to annotate each prompt with the structure-level constraint that fixes output format.

We seed the annotator with two output formats, JSON and Markdown (informed by the output format categories in ComplexBench (Wen et al., 2024)), and allow it to propose a new category whenever no existing one fits. New proposals are reviewed manually and either accepted as a new category or merged into an existing one. Beyond the free-form String case, which covers 50.8% of the corpus, the four most frequent categories are Labeled Sections (15.0%), JSON (14.0%), List (12.7%), and Markdown (8.9%), where each percentage is the share of prompts that include that category, and a prompt may declare multiple categories. We adopt these four formats together with String as the host formats of IFHierBench: the four structured formats define containers that recursively hold leaf-level value constraints, while String hosts value-level constraints directly without any container scope.

To confirm that hierarchical structure is not merely hypothetical, we measure the nesting depth of the format declaration in each prompt. Among the 606 format-bearing prompts, 411 (67.8%) carry at least one nested scope, with 255 at depth 1, 131 at depth 2, and 25 at depth 3 or deeper. Beyond depth, 171 of these prompts (28.2%) declare two or more formats, indicating that nesting in real prompts crosses format boundaries rather than recursing inside a single format. Constraint specifications also take up much of the prompt: across the 1,232 prompts they occupy 46.4% of the text on average. These findings shape IFHierBench: the five formats become its host scopes, nesting up to depth three makes each prompt a depth-stratified constraint tree, and the 28.2% of prompts that mix formats motivate letting a child scope take a different format from its parent.

4 IFHierBench Construction Pipeline

As shown in Figure 2, the IFHierBench construction pipeline runs in three stages. *Task setup preparation* parses each seed prompt into a clean task and its context together with a set of task-related keywords, and decides which structure-level constraints are admissible for the task. *Hierarchical sampling* first grows the constraint-tree skeleton from root structure-level constraints, then samples content-level templates at each leaf, and finally resolves parameters across all templates to finalize the tree. An automatic checker is derived from the finalized tree to verify model responses. *Prompt construction* renders the constraint tree into a single natural-language instruction, resolves cross-scope references, and merges constraints into more compact and readable sentences.

4.1 Task Setup Preparation

This stage turns each seed prompt into a sampling-ready task: a clean task statement free of output constraints, the context the task operates on, a task-specific vocabulary pool, and the subset of root formats that are admissible for the task's genre.

Task and Context Extraction. The first step parses each IFEval seed prompt into a task description the model must complete, together with the material it operates on. We use GPT-5.4 to return a clean task description with all output-related constraints stripped, together with the verbatim context the model operates on. The clean task lets downstream sampling attach new constraints to a goal that no longer carries any of its own, and the verbatim context keeps the rendered prompt self-sufficient under subsequent processing. Our running example (Figure 3, panel 1) illustrates this step: the seed *"Write a news brief of no more than 80 words based on the following article."* is parsed into the clean task *"Write a news brief based on the following article."* with its output constraints stripped, together with the article as the context.

Figure 3: running example for 3.1

Seed prompt
"Write a news brief of no more than 80 words based on the following article. [article body omitted]"

(1) Task and context
task: "Write a news brief based on the following article."
context: [article body omitted]

(2) Mined vocabulary pool
related keywords: report, policy, briefing, ...

(3) Genre and admissible root formats
genre: news brief
admits: JSON, Markdown, Labeled Sections, String

Keyword Mining. Since some constraints restrict the lexical choice of the output (e.g., *requiring the response to include specific keywords*, *forbidding specific keywoprd*, *naming a JSON key*), we use GPT-5.4 to mine a set of vocabulary pools tailored to each task. These pools feed the sampler in the next stage. Panel 2 of Figure 3 shows the pool emitted for the running example.

Genre-Format Mapping. Different tasks target different output genres (e.g., *news brief*, *poem*, *email*, *analysis report*), and each genre carries implicit restrictions on the admissible format: a news brief naturally accepts a JSON object, labeled sections, or markdown headings (Figure 3, panel 3), whereas an email rejects bulleted lists or JSON wrappers. For each task we ask GPT-5.4 to extract its genre and then to decide which of our predefined root formats are admissible given that genre, and the sampler draws a root format only from this admissible subset. Because task extraction, keyword mining, and genre-format mapping are each carried out by GPT-5.4, we randomly sample 10% of each step's outputs and manually evaluate them, and every step achieves over 90% accuracy.

4.2 Hierarchical Sampling

Given a prepared task, this stage produces a tree of constraints rooted at the chosen output format, together with the deterministic checker that will later score model responses against it.

Structure-Level Construction. The sampler walks the tree depth-first. At each scope it first chooses the shape of the scope (*string*, *JSON object*, *list*, *markdown document*, or *labeled section*) and the structure-level constraints that the scope as a whole must satisfy, such as a required JSON key set, a fixed list length, or a mandatory markdown heading. Container scopes recursively spawn child scopes whose kind is resolved through a weighted choice that respects the parent kind and the remaining depth budget, and at the maximum depth children are forced to be atomic strings to avoid empty containers. The running example (top panel of Figure 4) illustrates this: a JSON root carries required keys "headline" and "bullets", "bullets" expands into a length-3 list scope, and at the depth boundary each list element is forced to an atomic string. Each scope receives a unique alias when it is opened, so that constraints declared inside it can refer back to it unambiguously from the surface prompt.

Content-Level Sampling. Inside each scope, the sampler draws content-level constraints that further restrict the text produced at that scope, such as *required keywords*, *forbidden words*, *casing patterns*, or *length counts*. A candidate is admissible only if it does not conflict with siblings already drawn for the same scope, since two individually feasible constraints can be jointly unsatisfiable. Inspired by IFBench (Pyatkin et al., 2026), we maintain a *conflict matrix* that records pairwise incompatibilities between content constraints, and the sampler consults it on every draw. The matrix encodes physical mutex groups, in which at most one member may appear in any scope (for example, a scope cannot

simultaneously require its content to be *all uppercase* and *all lowercase*). In the running example (Figure 4, top panel), this stage commits keyword inclusion on "report" and "policy" plus a character cap at "headline", a list-length target at "bullets", and a word cap at each bullets[*].

Parameter Backfill. Each admitted constraint carries unresolved parameters, which the resolver binds in two stages.

Lexical resolution. Lexical parameters draw their values from the per-task vocabulary pool produced by Keyword Mining. In the example (Figure 4), the "headline" scope draws "report" and "policy" from the pool as its required keywords. Two consistency rules apply. First, since the required and forbidden words are drawn from the same pool, the sampler keeps them from colliding: no word is both required and forbidden, and neither is a substring of the other. Second, no keyword or identifier is drawn twice in the tree, so that every scope remains addressable by a unique alias.

Quantitative resolution. Let C, W, S, P denote the realized character, word, sentence, and paragraph counts of a scope, and write $X \in \{C, W, S, P\}$ with bounds $[X_{\min}, X_{\max}]$. Explicit constraints set one side at a time (e.g., *at least* N *words* sets $W_{\min} \leftarrow N$), or both sides for an interval. Each lexical commitment lifts the corresponding lower bound. Let T be the multiset of literal fragments forced by lexical commitments, where each required keyword contributes its literal and each pinned prefix, suffix, or postscript contributes its own. The resolver updates

$$C_{\min} \leftarrow \max\big(C_{\min}, \textstyle\sum_{t \in T} |t|\big),$$

$$W_{\min} \leftarrow \max\big(W_{\min}, \textstyle\sum_{t \in T} \mathrm{wc}(t)\big),$$

with $\mathrm{wc}(\cdot)$ denoting word count, and raises $S_{\min}$ and $P_{\min}$ by one for each pinned sentence- or paragraph-level fragment. Following readability studies (Shannon, 1951; Kincaid et al., 1975), the resolver also enforces inter-unit ratio bands

$$1 \le S/P \le 8, \;\; 3 \le W/S \le 40, \;\; 3 \le C/W \le 8,$$

which propagate bounds across units (e.g., $C_{\max} \le 8\, W_{\max}$ and $C_{\min} \ge 3\, W_{\min}$). The resolver iterates the floor updates and band propagation to a fixed point, rejecting the draw if any $X_{\min} > X_{\max}$. A final coarse-to-fine pass (paragraphs first, characters last) pins each count to a value inside its surviving $[X_{\min}, X_{\max}]$. On the example, the character cap on "headline" sets $C_{\max} = 60$, and the two required keywords lift $C_{\min}$ from 0 to 13, leaving the feasible interval $[13, 60]$.

Checker Generation. The checker mirrors the finalized constraint tree node by node. Each constraint type has a corresponding hand-written deterministic checker. At each node it runs a boolean test on the slice of the response that belongs to that scope, and a container passes only when every child also passes on its own slice. Locating that slice is itself a parse step: each node first runs an extractor that descends from its parent's text to the region it governs, such as parsing the response as a JSON object, indexing into a list, or isolating the section under a heading, and the node's boolean test runs only after this extraction succeeds. If the response cannot be parsed down to a node's scope, the node fails and its entire subtree fails with it, so a constraint is never evaluated against the wrong span of text. For the running example (Figure 4, bottom panel), the root parses the response as a JSON object and checks that its keys cover "headline" and "bullets". The "headline" value is then checked for the substrings "report" and "policy" and a 60-character cap, the "bullets" value is checked for list length 3, and each list item is checked for at most 15 words. Each prompt in the released benchmark ships with its own deterministic checker.

Figure 4: running example for 3.2

Annotated constraint tree

```
ROOT  (JSON object)
 +-- keys_required = {headline, bullets}
 +-- headline  (string, char_count <= 60)
 |     +-- must_include "report"
 |     +-- must_include "policy"
 +-- bullets  (list, list_length = 3)
       +-- bullets[*]  (string, word_count <= 15)
```

Per-node checker (one-line Python predicate)

```
root         : d = json.loads(x)
root.keys    : {"headline","bullets"} <= set(d)
headline     : "report" in s and "policy" in s
headline     : len(s) <= 60
bullets      : len(L) == 3
bullets[*]   : len(b.split()) <= 15
```

4.3 Prompt Construction

We convert the constraint tree into a single natural-language prompt by rendering each atomic constraint through a template and merging compatible sibling constraints into joint sentences.

Figure 5: running example for 3.3

Write a news brief based on the following article. Return a JSON object with the keys "headline" and "bullets". The headline must include "report" and "policy" and contain at most 60 characters. The bullets field is a list of 3 items, each at most 15 words.

[article context omitted]

Template Rendering. Template Rendering converts each leaf constraint into one sentence. Consider the "headline" node in the running example (Figure 4), which carries the constraint that the headline string must contain the keywords "report" and "policy". We pair every constraint type with a small set of templates, and the "required keywords" type maps to templates such as *"{alias} must contain {keywords}."*, where {alias} names the scope and {keywords} is the list of required words. Two slot values are then resolved. The alias is assigned by walking the tree top-down: the root is named "the response", and each child receives a phrase describing how it is reached from its parent (the value under a JSON key, an item at a position in a list, the section under a markdown heading). The JSON value under "headline" thus becomes "the headline", and the first element of the "bullets" list becomes "item 1 of bullets". The parameter slot is filled with the value committed during Parameter Backfill, here the list {report, policy}. Substituting both yields *"The headline must contain 'report' and 'policy'."*. The same procedure renders every other leaf in the example as a single sentence (e.g., *"The headline contains at most 60 characters."*).

Constraint Merging. Constraint Merging collapses sibling sentences in the same scope into a single, more readable sentence. In the running example, the "headline" scope holds two rendered sentences after the previous step, *"The headline must contain 'report' and 'policy'."* and *"The headline contains at most 60 characters."*. We predefine a library of compact merge templates, where each template targets a specific combination of sibling constraint types. When the constraints in a scope match such a template, we apply it with probability 0.8 and replace the separate sentences with the template's single sentence. Here the matched template rewrites the two sentences as *"The headline must include 'report' and 'policy' and contain at most 60 characters."*, the headline clause of the final prompt in Figure 5.

5 Experiments

5.1 Experimental Setup

5.1.1 Models

We evaluate seven models on IFHierBench, covering both leading proprietary APIs and open-weight checkpoints. Through hosted APIs we query gpt-5.5-2026-04-23 (Singh et al., 2025), claude-opus-4-6-20251015 (Anthropic, 2026), gemini-3-flash-preview (Google DeepMind, 2025), deepseek-r1 (Guo et al., 2025), and kimi-k2.5 (Team et al., 2026). The remaining two open-weight models, Qwen3.6-35B-Instruct (Yang et al., 2025) and gemma-4-26b-it (Google DeepMind, 2026), run through Ollama v0.21.0 on a single NVIDIA RTX 4090 (24 GB) using its default 4-bit quantization.

We set the thinking effort of GPT-5.5 and Claude Opus 4.6 to high, and keep the provider default temperature because these endpoints do not permit temperature adjustment. All other models decode at temperature 0 for deterministic outputs.

5.1.2 Evaluation Metrics

Following IFEval (Zhou et al., 2023), we report two strict accuracy metrics. Let N denote the number of prompts in the benchmark, and let C_i denote the set of constraints attached to prompt i. We write $\mathbf{1}[c] = 1$ if the Python checker for constraint c passes on the response and 0 otherwise.

Table 2: Prompt-level and instruction-level accuracy, overall and by depth.

Model	Prompt-Level Accuracy (%)					Instruction-Level Accuracy (%)				
	Overall	d=0	d=1	d=2	d=3	Overall	d=0	d=1	d=2	d=3
GPT-5.5	**53.7**	**86.7**	**49.3**	**43.3**	**35.3**	**65.3**	**95.2**	59.8	**59.7**	**66.9**
Claude Opus 4.6	42.8	78.7	42.0	28.0	22.7	61.9	93.0	**60.3**	56.7	61.1
Gemini-3-Flash	33.2	74.7	38.0	14.0	6.0	24.1	80.4	43.9	20.5	7.9
DeepSeek-R1	15.5	52.0	5.3	2.0	2.7	37.7	71.0	42.4	35.7	31.4
Kimi-K2.5	14.7	52.0	3.3	2.0	1.3	33.0	78.7	28.2	22.7	35.1
Qwen-3.6-35B	14.8	48.7	7.3	2.0	1.3	22.4	66.5	30.5	18.8	13.9
Gemma-4-26B	19.8	63.3	10.0	2.0	4.0	33.7	85.3	45.1	29.0	23.1

The **prompt-level accuracy** counts a response as correct only when every constraint is satisfied:

$$\mathrm{Acc}_{\mathrm{prompt}} = \frac{1}{N}\sum_{i=1}^{N}\prod_{c\in C_i}\mathbf{1}[c]. \tag{1}$$

The **instruction-level accuracy** averages over individual constraints rather than whole prompts:

$$\mathrm{Acc}_{\mathrm{inst}} = \frac{\sum_{i=1}^{N}\sum_{c\in C_i}\mathbf{1}[c]}{\sum_{i=1}^{N}|C_i|}. \tag{2}$$

5.2 Results

5.2.1 Overall Performance

Table 2 reports prompt-level and instruction-level strict accuracy on IFHierBench, overall and stratified by constraint-tree depth.

GPT-5.5 and Claude Opus 4.6 lead at every depth. GPT-5.5 and Claude Opus 4.6, both run with thinking effort set to high, top the table on every cell: 53.7% and 42.8% prompt-level overall, far ahead of Gemini-3-Flash at 33.2% and a tightly bunched cluster of Gemma-4-26B (19.8%), DeepSeek-R1 (15.5%), Qwen-3.6-35B (14.8%), and Kimi-K2.5 (14.7%). The ranking is preserved across d=0 through d=3, and the gap widens with depth. At d=3 GPT-5.5 and Claude Opus 4.6 retain 35.3% and 22.7% prompt-level accuracy, while every other model collapses to 6.0% or below.

Prompt-level accuracy decays steeply with depth. Every model loses between 37 and 53 absolute points from d=0 to d=1, including the strongest two (GPT-5.5 from 86.7% to 49.3%, Claude Opus 4.6 from 78.7% to 42.0%). This universal cliff at the very first level of nesting suggests that the prompts current models see during instruction-tuning are dominated by flat constraint structures, and that hierarchical instruction-following remains a uniformly underdeveloped capability across today's leading models. Accuracy continues to fall as depth grows, but GPT-5.5 and Claude Opus 4.6 retain non-trivial pass rates at d=3 (35.3% and 22.7%), while the other five collapse to 6.0% or below (Gemini-3-Flash 6.0%, Gemma-4-26B 4.0%, DeepSeek-R1 2.7%, Qwen-3.6-35B 1.3%, Kimi-K2.5 1.3%). The persistence at depth for the two strongest models most likely traces to their advantage in parameter count and training-data volume, which together support generalization to instruction-following regimes that are thinly represented during training.

The two metrics expose two distinct failure modes. Four of the five lower-performing models exhibit scattered content slippage: DeepSeek-R1, Gemma-4-26B, Kimi-K2.5, and Qwen-3.6-35B all show instruction-level accuracy well above prompt-level, with individual leaf constraints often passing but rarely all of them in the same response. Their leaf accuracy decays gradually with depth rather than collapsing, marking per-constraint slippage as the dominant error mode across these four. Gemini-3-Flash shows the opposite signature: instruction-level (24.1%) drops below prompt-level (33.2%), and leaf accuracy falls off a cliff with depth. A failure at an outer container marks every descendant leaf as not applicable, so when Gemini-3-Flash fails it tends to miss the structure entirely, dragging the entire subtree down. GPT-5.5 and Claude Opus 4.6 form a third pattern: the two metrics stay in a tight, positively correlated band, with instruction-level only 1.22× and 1.45× the prompt-level and leaf accuracy above 54% at every depth.

5.2.2 Failure Analysis

We look at failure from two complementary angles: which constraint types account for the largest share of failures, and how the two models that retain accuracy at d=3 plan their responses through the constraint hierarchy.

Table 3: Five hardest constraint types.

Model	string length	num words	nth para first word	value in set	num sentences
Pooled pass (%)	53.5	67.3	73.2	75.6	81.4
GPT-5.5	**90.1**	**96.7**	**97.8**	82.4	**98.3**
Claude Opus 4.6	68.2	87.2	96.4	**88.2**	95.7
Gemini-3-Flash	77.8	80.6	82.1	84.6	85.8
DeepSeek-R1	28.5	37.0	58.2	47.1	62.0
Kimi-K2.5	29.8	48.6	40.4	77.8	68.6
Qwen-3.6-35B	24.1	47.8	30.7	71.4	66.9
Gemma-4-26B	32.3	46.7	65.5	80.0	70.8

Table 3 ranks every constraint type by pooled pass rate across all seven models and reports each model's pass rate on the top five. The five hardest types are *string_length*, which fixes the response to an exact character count (53.5%), *num_words*, an exact word count (67.3%), *nth_paragraph_first_word*, the first word of the N-th paragraph (73.2%), *value_in_set*, which restricts a value to a fixed set (75.6%), and *num_sentences*, an exact sentence count (81.4%).

Four of the five are numeric in nature, namely counting characters, counting words, counting sentences, and indexing the N-th paragraph, and they dominate the difficulty profile. This is also where the gap between the three best models and the other four is widest. On *string_length*, GPT-5.5, Claude Opus 4.6, and Gemini-3-Flash pass 68% to 90% while the remaining four cluster at 24% to 32%. The pattern repeats on *num_words* (81% to 97% versus 37% to 49%) and on *nth_paragraph_first_word* (82% to 98% versus 31% to 66%). The hardest of the three is *string_length*, and one plausible explanation is exposure during instruction tuning: IFEval ships checkers for both word count and paragraph-indexed first word, but no checker for exact character length, so models likely receive far less training signal on character-precise length control.

Conversely, the two models that retain non-trivial accuracy at d=3, GPT-5.5 and Claude Opus 4.6, both run with thinking effort set to high, survive precisely because they reason explicitly about the constraint tree before responding. We examine their reasoning traces to ask whether this planning step walks the tree branch-by-branch (depth-first) or level-by-level (breadth-first). On the 150 depth-3 prompts the two models trace in different styles. GPT-5.5 writes topic-by-topic passages of the form "I am thinking about how to satisfy X", each focused on one specific constraint. Claude Opus 4.6 instead opens almost every trace by restating the prompt's constraints as a numbered list (1., 2., 3., ...) before turning to how to satisfy each one. Because this enumeration is a verbatim echo of the prompt, we strip it from each Claude Opus 4.6 trace. GPT-5.5 traces need no such preprocessing.

For each constraint we pick a short distinctive phrase from its specification, such as the keys, keywords, or numbers it names, and record where that phrase first appears in the trace. Ordering constraints by these first-mention positions gives the sequence in which the model brought each one into its reasoning, and we label each prompt by whether the depth-first or the breadth-first traversal of the constraint tree matches that sequence. Both models lean depth-first. Claude Opus 4.6 matches depth-first on 67% of prompts and breadth-first on 33%, while GPT-5.5 matches depth-first on 62% and breadth-first on 38%. The depth-first lead widens for both models on prompts where the two canonical orderings are themselves well separated, ruling out an accidental near-tie. Claude Opus 4.6's DFS lead over BFS is about 1.4$\times$ that of GPT-5.5, in line with the qualitative trace styles.

6 Conclusion

Existing instruction-following benchmarks treat constraints as a flat checklist, leaving the hierarchical structure of real-world prompts unevaluated. We address this with IFHierBench, a benchmark of 600 prompts stratified across four constraint depths and built from 35 programmatically checkable constraint types. Evaluating seven leading models, we find that they handle flat (d=0) constraints well, but accuracy drops sharply as nesting deepens, with the strongest model only marginally exceeding 50%. Hierarchical instruction following remains an open capability gap for today's strongest LLMs.

AI USE STATEMENT

In accordance with the ICLR policy, we used generative AI tools for one task with required disclosure: an LLM runs the analysis stages of the benchmark construction pipeline, exactly as described in the main text. We additionally used large language models as writing aids for improving grammar, fluency, and stylistic consistency, rephrasing sentences for clarity, and polishing LaTeX table code. The pipeline outputs are audited by manual spot checks and deterministic checkers, no passage was adopted without manual review and editing by the authors, and we take full responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

Our code and data are available at `https://anonymous.4open.science/r/IFHierBench-0087`. The appendix reproduces the full prompts used by the two LLM-run analysis stages, the complete constraint inventory, and example prompts at every depth. The released benchmark ships every prompt with its deterministic checker and the evaluation protocol of the experiments section.

REFERENCES

Anthropic. Introducing claude opus 4.6. `https://www.anthropic.com/news/claude-opus-4-6`, 2026. Accessed: 2026-05-19.

Ge Bai, Jie Liu, Xingyuan Bu, Yancheng He, Jiaheng Liu, Zhanhui Zhou, Zhuoran Lin, Wenbo Su, Tiezheng Ge, Bo Zheng, et al. Mt-bench-101: A fine-grained benchmark for evaluating large language models in multi-turn dialogues. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7421–7454, 2024.

Sumanth Dathathri, Andrea Madotto, Janice Lan, Jane Hung, Eric Frank, Piero Molino, Jason Yosinski, and Rosanne Liu. Plug and play language models: A simple approach to controlled text generation. *arXiv preprint arXiv:1912.02164*, 2019.

Google DeepMind. Gemini 3 Flash. `https://deepmind.google/technologies/gemini/`, 2025. Accessed: 2026-05-19.

Google DeepMind. Gemma 4. `https://deepmind.google/models/gemma/gemma-4/`, 2026. Accessed: 2026-05-19.

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.

Chris Hokamp and Qun Liu. Lexically constrained decoding for sequence generation using grid beam search. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1535–1546, 2017.

Yuxin Jiang, Yufei Wang, Xingshan Zeng, Wanjun Zhong, Liangyou Li, Fei Mi, Lifeng Shang, Xin Jiang, Qun Liu, and Wei Wang. Followbench: A multi-level fine-grained constraints following benchmark for large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4667–4688, 2024.

Nitish Shirish Keskar, Bryan McCann, Lav R Varshney, Caiming Xiong, and Richard Socher. Ctrl: A conditional transformer language model for controllable generation. *arXiv preprint arXiv:1909.05858*, 2019.

Maxim Khanov, Jirayu Burapacheep, and Yixuan Li. Args: Alignment as reward-guided search. *arXiv preprint arXiv:2402.01694*, 2024.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406*, 2022.

J Peter Kincaid, Robert P Fishburne Jr, Richard L Rogers, and Brad S Chissom. Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel. Technical report, 1975.

Ben Krause, Akhilesh Deepak Gotmare, Bryan McCann, Nitish Shirish Keskar, Shafiq Joty, Richard Socher, and Nazneen Fatema Rajani. Gedi: Generative discriminator guided sequence generation. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pp. 4929–4952, 2021.

Wai-Chung Kwan, Xingshan Zeng, Yuxin Jiang, Yufei Wang, Liangyou Li, Lifeng Shang, Xin Jiang, Qun Liu, and Kam-Fai Wong. Mt-eval: A multi-turn capabilities evaluation benchmark for large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 20153–20177, 2024.

Jinnan Li, Jinzhe Li, Yue Wang, Yi Chang, and Yuan Wu. Structflowbench: A structured flow benchmark for multi-turn instruction following. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 9322–9341, 2025.

Xun Liang, Hanyu Wang, Yezhaohui Wang, Shichao Song, Jiawei Yang, Simin Niu, Jie Hu, Dan Liu, Shunyu Yao, Feiyu Xiong, et al. Controllable text generation for large language models: A survey. *arXiv preprint arXiv:2408.12599*, 2024.

Michael Xieyang Liu, Frederick Liu, Alexander J Fiannaca, Terry Koo, Lucas Dixon, Michael Terry, and Carrie J Cai. " we need structured output": Towards user-centered constraints on large language model output. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*, pp. 1–9, 2024.

Ximing Lu, Peter West, Rowan Zellers, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Neurologic decoding:(un) supervised neural text generation with predicate logic constraints. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4288–4299, 2021.

Yuetian Mao, Junjie He, and Chunyang Chen. From prompts to templates: A systematic prompt template analysis for real-world llmapps. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pp. 75–86, 2025.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Kaiser Pister, Dhruba Jyoti Paul, Ishan Joshi, and Patrick Brophy. Promptset: A programmer's prompting dataset. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, pp. 62–69, 2024.

Matt Post and David Vilar. Fast lexically constrained decoding with dynamic beam allocation for neural machine translation. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pp. 1314–1324, 2018.

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.

Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, et al. The prompt report: a systematic survey of prompt engineering techniques. *arXiv preprint arXiv:2406.06608*, 2024.

Claude E Shannon. Prediction and entropy of printed english. *Bell system technical journal*, 30(1): 50–64, 1951.

Seungmin Shin, Dooyoung Kim, and Youngjoong Ko. Eco decoding: Entropy-based control for controllability and fluency in controllable dialogue generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 28297–28309, 2025.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Y Charles, HS Che, Cheng Chen, Guanduo Chen, et al. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

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.

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.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

Kevin Yang and Dan Klein. Fudge: Controlled text generation with future discriminators. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 3511–3535, 2021.

Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, et al. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*, 2025.

Tao Zhang, Chenglin Zhu, Yanjun Shen, Wenjing Luo, Yan Zhang, Hao Liang, Fan Yang, Mingan Lin, Yujing Qiao, Weipeng Chen, et al. Cfbench: A comprehensive constraints-following benchmark for llms. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 32926–32944, 2025.

Yanjie Zhao, Xinyi Hou, Shenao Wang, and Haoyu Wang. Llm app store analysis: A vision and roadmap. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–25, 2025.

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.

A Appendix

A.1 Prompt Template

Two stages of the IFHierBench construction pipeline use LLM for analysis. We list both prompts in full below.

Format-type discovery. Before we fix the structural constraint inventory of IFHierBench, we run a corpus-wide analysis to identify which output formats real instruction-following prompts already request. For every prompt in the source corpus, the model classifies its required output formats against a dynamically updated category list, extracts the source sentence that introduces each format, and proposes new categories when none of the known ones fit. The model additionally returns the nested structure of these format constraints as a tree, where a child constraint targets a sub-part introduced by its parent (for example, a JSON-with-key parent and a value-shape constraint on that key as its child), providing direct evidence that real prompts already specify formats hierarchically. The format families discovered here seed the root-shape choices used by the constraint sampler. The full prompt is shown in Figure 6.

Format-type discovery prompt

```
=== SYSTEM ===

You are an expert at analyzing output format requirements in LLM prompt templates.

**Definition of "format":** A format is the structural data format of the output -- how the output is organized and structured. Examples: JSON, Markdown, etc.
Format is about the STRUCTURE of the output, NOT about the content. The following are NOT formats:
- Content constraints (e.g., "must include keyword X", "output must be a single word yes/no")
- String matching constraints (e.g., "use the exact phrase '...'")
- Tone/style constraints (e.g., "be concise", "use formal language")
- Numerical constraints (e.g., "under 100 words") unless they describe structural elements
- Fixed/canned responses (e.g., "output 'N/A' if not applicable")

Given a prompt template, your task is to:
1. Identify ALL output FORMAT types required by this prompt (structural formats only)
2. Classify each format into one of the known categories, OR propose a new category if none fits
3. Extract the EXACT sentence(s) from the prompt that define each format requirement
4. Analyze the hierarchical/nested structure of the format constraints as a tree:
   - Each constraint that operates on a sub-part defined by a parent constraint is a child node
   - There are NO predefined semantics for each level -- any constraint type can appear at any level
   - The nesting relationship is purely structural: a child constraint targets something defined/created by its parent constraint

Current known format categories (dynamically updated):
{categories_json}

Rules:
- "Format" means structural data format ONLY. Do not classify content-level constraints (string matching, fixed phrases, tone, word count, etc.) as formats.
- First check if the format fits an existing category. Only propose a new category if none fits.
- A prompt may have zero, one, or multiple format requirements.
- "No format requirement" is a valid answer if the prompt does not constrain output format. Many prompts have content constraints but no structural format constraint
-- mark these as no_format_requirement = true.
- When proposing a new category, provide: name, description, and why existing categories don't fit. The new category must describe a structural data format, not a
content constraint.
- For hierarchy analysis: represent constraints as a tree. A constraint is a child of another constraint if it applies to a sub-structure that the parent constraint defines. For
example, if a parent says "output JSON with key 'items'", and a child says "'items' must be a list", the child is nested under the parent because it targets 'items' which
the parent introduced. A child of that could be "each element in 'items' must have key 'name'" -- again targeting a sub-part of the parent. There is no limit on depth,
and any constraint type (format, numerical, string, etc.) can appear at any level as long as it targets a sub-structure.

=== USER ===

Analyze the format requirements in the following complete prompt template. Focus on the FULL prompt text as the primary source of truth. The auxiliary annotations
below are provided only as reference and may be incomplete.

---PROMPT START---
{prompt_text}
---PROMPT END---

[Auxiliary] Output Format/Style annotation: {output_format_style}
[Auxiliary] Extracted constraints (JSON): {extracted_constraints}

Return your analysis using the report_format_analysis function.
```

Figure 6: Prompt for format-type discovery.

Task preparation. For each base prompt drawn from the source corpus, we issue a single LLM call that (i) strips every pre-existing format, length, count, and style instruction, leaving only a bare content goal in the `task` field, (ii) classifies the task into one of twelve genres, (iii) proposes domain-specific vocabulary (`content_hints`) that downstream samplers reuse as constraint parameters, (iv) detects and copies out any inline context the task operates on (such as a sentence to rewrite or a passage to summarise), (v) emits an audit list of every constraint string removed under step (i) so the stripping pass is itself verifiable, and (vi) records the subset of root output shapes that would feel natural for the cleaned task. The cleaned task and content hints become the inputs to the constraint-tree sampler in the next stage. The full prompt is shown in Figure 7.

A.2 Constraint Inventory

Tables 4 and 5 list all 35 constraints in IFHierBench: the 10 structure-level constraints that operate on container scopes (JSON, Markdown, list, and labeled-section regions) and the 25 content-level constraints that operate on string scopes. Each entry shows the constraint identifier, a short definition of what the deterministic Python checker verifies, and one of the natural-language templates from the prompt-construction stage.

Table 4: Constraint inventory of IFHierBench (Part 1 of 2). Lists the 10 structure-level constraints and the first 6 of 25 content-level constraints. The remaining 19 content-level constraints are listed in Table 5.

Level	Constraint	Short definition	Example NL template
Struct.	json.has_keys	The JSON object must contain at least the listed keys at its top level.	The response must include at least the following keys: {keys_joined}.
	json.key_count	The number of top-level keys in the JSON object satisfies a relation against a target count.	The response must have {rel} {n} top-level keys.
	array.length	The number of elements in the array satisfies a relation against a target count.	The response must have {rel} {n} elements.
	array.elements_at	Items at specified 0-indexed positions of the array are scoped as nested sub-shapes that carry further child constraints.	Items at positions {positions_joined} of the response must satisfy the constraints below.
	list.items_unique	The list contains no duplicate items.	All items in the response must be unique.
	markdown.heading_count	The Markdown document contains a target number of headings, optionally restricted to one heading level.	The response must include {rel} {n} headings{level_suffix}.
	markdown.has_heading	The Markdown document contains a listed set of named headings (each opening a child subscope).	The response must include the following headings: {headings_joined}.
	markdown.has_code_block	The Markdown document contains at least one fenced code block of the given language, with no comments inside the code.	The response must contain at least one {language} code block. The code inside must not contain any comments.
	labeled.has_labels	The response contains a listed set of labeled sections, each label at line start followed by a colon.	The response must include at least the labels: {labels_joined}, with each label placed at the start of a line followed by a colon.
	labeled.section_count	The response contains a target number of labeled sections under the same colon-suffixed convention.	The response must contain {rel} {n} labeled sections. Mark each section with a label followed by a colon at the start of a line.
Content	string.string_length	The response length in characters falls within an inclusive [min, max] range.	The response must be between {min} and {max} characters long.
	string.num_words	The number of words in the response satisfies a relation against a target count.	The response must contain {rel} {n} words.
	string.num_sentences	The number of sentences in the response satisfies a relation against a target count.	The response must contain {rel} {n} sentences.
	string.nth_paragraph_first_word	The response has exactly n paragraphs and the i-th paragraph begins with a fixed word.	The response must have exactly {n} paragraphs; the {nth_ord} paragraph must begin with the word "{first_word}".
	string.contains	The response contains every item in a list of required substrings.	The response must contain the text: {words_joined}.
	string.not_contains	The response contains none of a list of forbidden substrings.	The response must not contain any of: {words_joined}.

Table 5: Constraint inventory of IFHierBench (Part 2 of 2). Lists the remaining 19 of 25 content-level constraints. The first 6 content-level constraints and all 10 structure-level constraints are listed in Table 4.

Level	Constraint	Short definition	Example NL template
	string.starts_with	The response begins with a fixed prefix string.	The response must start with "{prefix}".
	string.end_checker	The response ends with an exact phrase, with nothing after it.	The response must end with the exact phrase "{end_phrase}". No other words or characters should follow this phrase.
	string.keyword_ frequency	Each listed keyword occurs in the response with a specified frequency.	The response must satisfy the following word counts: {specs_joined}.
	string.letter_frequency	Each listed letter occurs in the response with a specified frequency.	The response must contain the letters as specified: {specs_joined}.
	string.capital_word_ frequency	The number of fully capitalized words in the response satisfies a relation against a target count.	The response must contain {rel} {n} fully capitalized words.
	string.capital_letters	The response is written entirely in uppercase letters.	The response must be entirely in capital letters.
	string.lowercase_letters	The response is written entirely in lowercase letters.	The response must be entirely in lowercase.
	string.comma	The response contains no commas.	The response must not contain any commas.
	string.quotation	The response is wrapped entirely in double quotes.	The response must be wrapped entirely in double quotes.
Content	string.title	The response includes a designated title wrapped in double angle brackets.	The response must include the title «{title_text}» wrapped in double angle brackets.
	string.bullet_list	The response contains exactly n bullet points (using * or -).	The response must contain exactly {n} bullet points (using * or -).
	string.section_checker	The response is split into n sections, each beginning with a fixed literal splitter.	The response must have {n} sections. Mark the beginning of each section with '{splitter}'.
	string.highlight_section	The response contains at least n inline highlighted passages wrapped in asterisks.	The response must contain at least {n} highlighted passages wrapped in asterisks, like *this*.
	string.placeholder	The response contains at least n bracketed placeholders of the form [NAME].	The response must contain at least {n} placeholders in square brackets, like [NAME].
	string.postscript	The response ends with a postscript opened by a fixed marker and followed by exactly one sentence.	The response must end with a postscript starting with '{marker}', followed by exactly one sentence and nothing else.
	string.value_in_set	The (sub-)value at this scope must equal one of an allowed value list.	The response must be one of: {values_ joined}.
	string.constrained_ response	The whole response must be exactly one of an allowed value list.	Your answer must be exactly one of: {values_joined}.
	string.two_responses	The response is two distinct outputs separated by the literal ******.	Produce two distinct responses separated by ******.
	string.repeat_prompt	The response first repeats the prompt verbatim and then answers it.	First repeat the prompt verbatim, then answer it.

A.3 Example Prompts by Constraint Depth

We show one representative IFHierBench prompt at each constraint depth, from a single flat constraint (d=0) to three levels of nested scopes (d=3).

Depth 0

Write a poem about a curious cat.
Write the response in lowercase only.

Depth 1

Write a rap about an abyss.
Respond using Markdown formatting.
Include these top-level headings: "Imagery".
The section under heading 'Imagery' must be entirely in capital letters.
The response must include at most 3 headings.

Depth 2

Recommend a college with open enrollment and regional accreditation.
Respond with a JSON object.
The response must include at least the following keys: "transfer_credits".
The value of key 'transfer_credits' must include the following headings: "Pros and Cons".
The section under heading 'Pros and Cons' must be entirely in capital letters.

Depth 3

Write a riddle that describes the word 'key'.
Respond with a JSON object.
The response must include at least the following keys: "target_audience".
The value of key 'target_audience' must contain at least the keys "Sources".
The following positions in the value of key 'Sources' have specific requirements: 2.
Write item 2 of the value of key 'Sources' in lowercase only.
All items in the value of key 'Sources' must be unique.
Include exactly 3 entries in the value of key 'Sources'.

Task preparation prompt

```
=== SYSTEM ===

You annotate prompts for a constraint-following benchmark.

Your job for EACH input prompt:
1. Strip from the prompt EVERY token that pins the response's shape, length, count, format, medium, case, punctuation, or scaffold layout, leaving ONLY a bare content goal in `task`. Specifically remove:
  * EVERY explicit count or number that bounds output structure ("five Q&A pairs", "two paragraphs", "exactly 3 sections", "at least 100 words", "in 200 characters").
  * EVERY format / medium / structure word ("as JSON", "in HTML", "Markdown table", "bulleted list", "with headings", "in paragraphs", "wrapped in quotes", "all caps", "in lowercase", "with the title in bold").
  * EVERY scaffold / separator template / format example ("separated by ***", "using the format: Q: ... A: ...", "indicated by: * example bullet").
  * ALSO strip format names even when they appear as content nouns ("JSON schema", "HTML page", "Markdown document", "XML file", "CSV table", "YAML config", "SQL query"). Rephrase the task so the content goal stands alone WITHOUT naming any output format:
     "Write a JSON schema for a user profile with name, email, age."
      -> "Describe a user profile structure with name, email, age."
     "Generate an HTML page advertising a camera."
      -> "Write an advertisement for a camera."
     "Produce a Markdown tutorial on sorting."
      -> "Explain how to sort a list."
    The format intent is carried entirely by `allowed_root_formats` below; `task` must never name a format, in any role (directive OR content noun).
  The cleaned `task` reads like a plain content goal:
    "Provide question-and-answer pairs."
    "Write about banana peel biology."
    "List limericks about people named Bill."
  Keep genre / content-type nouns that are NOT format names (e.g., "advertisement", "poem", "recipe", "limerick", "email", "story") -- these describe what KIND of writing, not how it is rendered.
2. Classify the task into exactly one of 12 genres.
3. Propose domain-specific vocabulary (content_hints) that a constraint generator could plug into parameters like JSON key names, markdown heading titles, keyword requirements, etc. Hints must be semantically relevant to the task. ALL structural pools are MANDATORY (non-empty) regardless of genre -- the benchmark wraps every task in JSON / Markdown / Labeled root variants for hierarchical testing, so empty pools kill entire benchmark branches.
4. Detect inline `context`: material EMBEDDED IN THE PROMPT that the task operates ON, e.g., a sentence to rewrite, a passage to summarize, an email to reply to, a code snippet to refactor, a data row to describe, a list of items to reformat. If present, copy it VERBATIM into context.text (preserve quotes, line breaks, whitespace), set context.present=true, and pick the correct context.kind.

   `context` is ONLY text-to-be-processed by the model. The following are NOT context and must NOT be extracted as one:
  * A scaffold / format example / placeholder template ("* This is an example bullet", "Q&A #1 *** Q&A #2", "{ key: <value>, ... }" skeleton).
  * Format / structure instructions ("Put the names in bullet points", "Respond in JSON").
  * A repetition or rephrasing of the task itself.
  If the only embedded material is scaffold / instructions / task repetition, set context.present=false, text='', kind='none'. Self-contained generative prompts ('Write a poem about California', 'Suggest startup ideas') also have no context.
5. List each removed constraint verbatim for audit.
6. Decide `allowed_root_formats`: the subset of {string, json, list, markdown, labeled} that would feel natural as the response shape for THIS task. Use these definitions:
  * "string" -- free prose, no enforced structure. Natural for poems, lyrics, short open-ended answers, single-sentence rewrites, emails written as natural correspondence, stories.
  * "markdown" -- sectioned document with headings + paragraphs. Natural for essays, tutorials, articles, multi-section answers, anything that benefits from headings.
  * "list" -- the response is fundamentally an enumeration of items (rendered as a JSON array). Natural when the answer IS a list of options / suggestions / names / steps / ingredients.
  * "json" -- structured data with named fields. Natural for data records, product specs, schemas, comparisons with multiple attributes, anything where keyed access matters.
  * "labeled" -- sections written as "Label: content". Natural for recipes (Ingredients / Instructions), structured Q&A, parts of an analysis (Pros / Cons / Verdict).

  EXCLUDE any format that would feel absurd for this prompt's content. A poem must NOT allow "json" or "labeled". An email reply must NOT allow "list" or "json". A pure data-extraction task probably EXCLUDES "string". Include at least 2 formats when reasonable so the benchmark has variety, but never include a format that produces nonsense.

  SPECIAL CASE -- content intrinsically tied to one format: if the original prompt described content that IS a specific format (e.g., "Write a JSON schema...", "Generate an HTML page...", "Produce a Markdown tutorial..."), the format name has been stripped from `task` per rule 1, but the format intent must be preserved here. Set `allowed_root_formats` to that single format only (e.g., `["json"]` for the schema task, `["markdown"]` for the HTML / Markdown tutorial). Overriding the "at least 2 formats" guideline is correct in this case.

Rules:
- `task` must be a standalone, well-formed instruction -- the benchmark will append NEW formatting / count / structural constraints onto it, so it must not dictate any of those itself. Reject any task that still contains "exactly N", "as JSON", "in bullet points", "wrapped in quotes", "in paragraphs", etc.
- The `task` and the `context` are SEPARATE: `task` describes what to do (e.g., 'Rewrite the sentence in an unusual style.'); `context` holds the material to do it on. Do not concatenate them, do not paraphrase the context, and never put a scaffold / format example into context -- only real text-to-be-processed.
- Pool sizes (larger is better -- the benchmark samples without replacement). ALL pools below are MANDATORY for every genre.
   * suggested_keywords: 50-100 items (cover common terms + specific vocabulary; benchmark samples without replacement across ALL constraints)
   * suggested_json_keys: 10-25 items (snake_case; even prose tasks can be wrapped in JSON like {'title','intro','body','characters','theme',...})
   * suggested_headings: 10-20 items (any task can be sectioned; for a poem use ['Title','Opening Stanza','Imagery','Symbolism','Theme',...])
   * suggested_labels: 6-15 items (even casual answers can be grouped into ['Direct Answer','Reasoning','Caveats','Sources',...])
   * suggested_forbidden: exactly 2-3 items (keep TIGHT, high-frequency words)
- For `creative_writing` and `qa_answer`: do NOT skip structural pools. Imagine how a structured rewrite of the response would look and populate the keys / labels / headings accordingly. Empty pools are a bug.

=== USER ===

Prompt to annotate:

"""
{prompt}
"""

Return ONLY valid JSON matching the schema.
```

Figure 7: Prompt for task preparation.