Boosting Instruction Following at Scale

Anonymous ACL submission

Abstract

There are few control points a developer can leverage to control the behavior of an LLM inside an application. A common black-box approach to influence LLM inference is through careful manipulation of the prompt, such as by adding or modifying instructions. However, merely adding more instructions provides little assurance that they will actually be followed. This is especially true as the number of additional instructions grows. We introduce Instruction Boosting as a post-generation method to increase the reliability of LLM prompt instructions. We show that Instruction Boosting improves the instruction following rate by up to 7 points when boosting two instructions and up to 4 points when boosting ten instructions. To demonstrate these results we introduce SCALEDIF, a benchmark with a scaled instruction volume of up to ten instructions per data sample. We also present an analysis of the commonly observed trend that performance degrades as more instructions are added. We show that an important factor contributing to this trend is the degree of tension and conflict that arises as the number of instructions is increased. We contribute a quantitative conflict scoring tool that explains the observed performance trends and provides feedback to developers on the potential impact that additional prompt instructions have on a model's performance.

1 Introduction

Large Language Models (LLMs) have become foundational components in the development of agentic applications. However, for developers, these powerful models often behave like black boxes, making it difficult to precisely control their output. A typical method for influencing an LLM's behavior is through careful manipulation of the prompt, such as by adding or modifying instructions. For instance, when testing reveals an undesirable model outcome, a developer might add a corrective instruction to the prompt in an effort to prevent that behavior from recurring.

This reliance on prompt-based instructions, however, presents two fundamental challenges. First, there is little guarantee that a newly added instruction in the prompt will actually be followed by the LLM. Second, the progressive addition of instructions can inadvertently introduce tension or even direct contradictions with pre-existing instructions, making it more difficult to satisfy all instructions simultaneously. These issues combined can lead to an often observed phenomenon, where the instruction following rate (IF rate) degrades as the number of instructions increases (Jiang et al., 2024).

We propose *Instruction Boosting* as a test-time post generation method to increase the reliability of LLM instruction following in applications. Instruction boosting is predicated on the observation that it is often easier for an LLM to revise a suboptimal response to meet a set of instructions than it is to generate a perfect response in the first place. As such, our method borrows from concepts of self-correction (Madaan et al., 2023; Huang et al., 2024), employing techniques to refine and reinforce initial model responses against the given instructions. As shown in Fig. 1, we apply boosting on an initial response from an LLM in order to increase the number of instructions that are followed. We evaluate several boosting strategies and show instruction following improvements across a range of open LLMs. Specifically, we show that boosting improves the IF rate by up to 7 percentage points in the case of two prompt instructions and by up to 4 percentage points for ten instruction scenarios.

To rigorously evaluate our approach, we contribute SCALEDIF, a new instruction following benchmark that extends the popular IFEval dataset (Zhou et al., 2023) to include data samples with up to ten instructions. Our experimental results with SCALEDIF also independently confirm the typical performance degradation with increas-

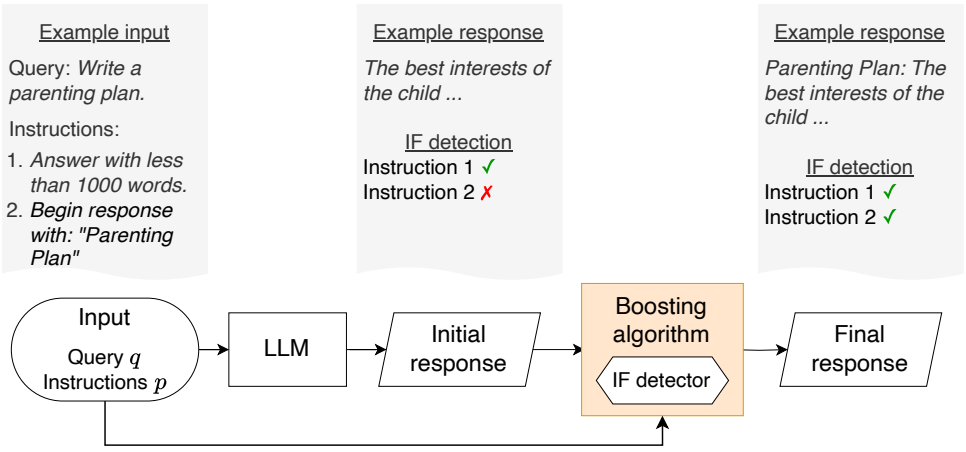

Figure 1: Overview of the instruction boosting approach.

ing numbers of instructions. Since each instruction constrains the model's response, adding more instructions can easily create an over-constrained problem. While some instruction sets may contain explicit conflicts—pairs of instructions that are impossible to follow simultaneously—we also show that even in the absence of such hard conflicts, a growing number of constraints can lead to tension between instructions, making it increasingly difficult to satisfy all of them at once.

We formalize this notion by defining a *soft conflict* as a pair of instructions that are difficult, though not impossible, to follow simultaneously. We developed a quantitative conflict scoring test that determines the degree of soft conflict among a set of instructions. We show that conflict scores are negatively correlated with both the initial IF rate and the improved IF rate after boosting.

We exploit this relationship by proposing the conflict scoring test as a valuable feedback tool for developers. Developers can compute conflict scores before and after adding additional instructions to a prompt to obtain crucial feedback about the impact the additional instructions have on model performance. The conflict scoring test can serve as an early indicator of expected instruction following performance and can guide developers in adjusting or modifying instructions to lower the conflict score, thereby obtaining improved responses and better overall model control.

This paper makes the following contributions:

1. The SCALEDIF dataset with an instruction volume of up to 10 instructions per data sample,

2. Instruction Boosting as a test-time method for developers to increase reliance on prompt-based instructions and control over LLM responses, and

3. An instruction conflict scoring test that estimates the complexity of satisfying a set of instructions simultaneously to provide feedback to developers.

2 SCALEDIF Dataset

To study the effects that additional instructions have on the overall IF rate, we created SCALEDIF, a derivative of the popular IFEval (Zhou et al., 2023) instruction following dataset with up to ten instructions per data sample. The original IFEval dataset contains 541 samples, where each sample contains a query and between one and three verifiable instructions that must be followed. There are 26 distinct classes of instructions, each with its own Python verifier function to validate if a given text follows the instruction.

An instruction is defined by a tuple consisting of a (i) textual *description*, (ii) a *class id* refering to an associated verifier function, and (iii) a set of verifier function *parameters* matching the instruction description, as illustrated below:

Description	"Answer with at least 50 words."
Class id	*length_constraints:number_words*
Parameters	{ relation: "at least", num_words: 50 }

The adherence of an LLM response to an instruction can be verified by invoking the associated verifier function on the response text with the given parameters.

SCALEDIF: Our derivative dataset consists of 538 of the 541 IFEval data samples, each containing a query (e.g., *'Can you elaborate on "I froze*

when I was jogging"?'), and ten unique instructions. We rewrote the IFEval samples to isolate the query from existing instructions so that the number of instructions added to the query could easily be varied. We used Mistral Large (Mistral AI, 2024a) for an initial extraction of the query from the original IFEval samples and then reviewed and refined the extracted queries manually.

We added ten unique instructions to each query to form the new SCALEDIF samples. We leverage the fact that the 26 instruction classes are largely independent of the (rewritten) queries. Any query can be paired with instructions from any unique instruction class. We made some changes to the instructions to ensure query-independence: we replaced three instruction classes (*response_language, english_uppercase, english_lowercase*) with three new query-independent ones (*length_constraints:sentence_length, detectable_format:yaml_format, startend:start_checker)*. This resulted in a set $\mathcal{P}$ of 26 query-independent instruction classes grouped into eight instruction categories: change case, combination, detectable content, detectable format, keywords, length constraints, punctuation, and startend.

Each of the 538 samples was then assigned a set of $N = 10$ instructions drawn from $\mathcal{P}$. To ensure diverse parameter values across instructions, we associated each instruction class $p \in \mathcal{P}$ with a sampling function S which samples values for its parameters, if any. For numerical parameters such as *number of words* or *number of paragraphs*, we sample parameter values from a simple 1-dimensional distributions. For parameters requiring keywords (e.g., *'don't use the word "hot"'*), keywords are sampled using an LLM (Granite 3.1-8B (IBM Granite, 2024)) in order to generate keywords that are relevant and meaningful in the context of a given query. This keyword sampling approach is in contrast to the original IFEval selection method which chose keywords uniformly at random from a large vocabulary.

Fig. 2 shows the number of instructions per instruction category in SCALEDIF. A full list of the instruction classes and representative examples is included in the Appendix.

Sampling instruction parameters independently for each the N instructions can easily lead to contradictions. For example, the *keywords:existence* instruction class requires that a set of keywords be present in the response, and the *keywords:forbidden_words* instruction class requires that a set of keywords not appear. These parameters must be sampled to be disjoint to avoid contradictory instructions that are impossible to follow simultaneously.

To solve this problem we enforced pairwise constraints during parameter sampling. When constructing an instruction set L and a candidate instruction I is selected to be added to L, first constraints C_I are computed based on the instructions already chosen for L. For example, if an instruction stating that the response must contain at least N paragraphs is already in L, then a constraint is added to ensure that no other instruction can require the number of words to be less than $N * 10$. These constraints are enforced while sampling pa-

Algorithm 1 Constrained Instruction Sampling

Require: Integer $N > 0$, set of Instructions $\mathcal{P}$
Ensure: List L of N valid (Instruction, Parameters) samples

$L \leftarrow$ empty list
while length$(L) < N$ **do**
  Sample p uniformly without replacement from $\mathcal{P}$
  Compute constraints C_p on p from instructions in L
  Sample required arguments A_p using strategy $P(C_p)$
  if a valid A_p can be found **then**
    Append (p, A_p) to L
  end if
end while
return L

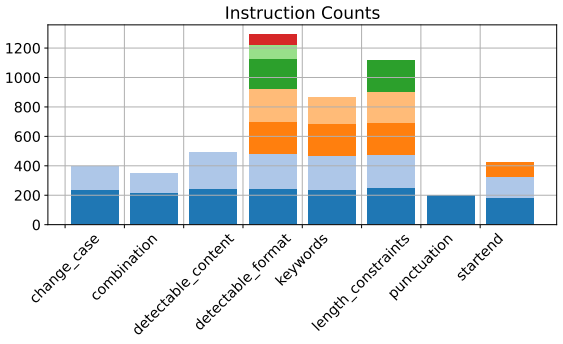

Figure 2: Number of instructions in SCALEDIF across eight instruction categories. The contribution of individual instruction classes in each category is distinguished by color.

rameters for the candidate instructions, and if they cannot be satisfied, then that instruction is rejected and a new candidate is sampled. If instruction parameters are successfully sampled which satisfy all existing constraints, then the candidate instruction is added to the list L. This process is described in Alg. 1.

After sampling ten instructions for each of the 538 samples, we shuffled the instruction order to avoid any sensitivity to the order in which instructions appear in the prompt. Finally, we constructed scaled down versions of the dataset with 2, 4, 6 and 8 instructions per sample by randomly removing instructions from the 10-instruction dataset version.

3 Instruction Boosting

The idea behind instruction boosting is to scale compute (Snell et al., 2024) in order to improve on the baseline IF rate. As illustrated in Fig. 1, instruction boosting operates as a post-generation step that, similar to self-correction (Madaan et al., 2023; Huang et al., 2024), employs techniques to refine and reinforce an initial model response against the given instructions. Instruction boosting can be enabled for a subset or all the instructions in given prompt. We devised several boosting strategies with different cost-performance trade-offs to give developers a choice when balancing costs and benefits.

We evaluated instruction boosting on SCALEDIF with several open models of various sizes including Llama-3.3-70B-Instruct[1] (Llama-70B), Llama-3.1-8B-Instruct[2] (Llama-8B), Qwen2.5-72B-Instruct[3] (Qwen-72B), Mixtral-8x7B-Instruct-v0.1[4] (Mixtral-8x7B) and Mixtral-8x22B-Instruct-v0.1[5] (Mixtral-8x22B).

As shown in Fig. 3 (solid lines), the initial IF rate that these models achieve on SCALEDIF ranges from 0.56 (Mixtral-7x8B) up to 0.88 (Llama-70B) with two instructions, and reduces to 0.39 (Mixtral-7x8B) and 0.66 (Llama-70B) with ten instructions. Fig. 3 also shows the instruction following boost (dotted lines) achieved with the best performance boosting strategy (Best-of-N), which will be explained in detail in the following section. With two instructions the largest IF rate boost is achieved by Mixtral-7x22B at 7 percentage points. With ten instructions, the largest boost is by Llama-70B at 4 percentage points.

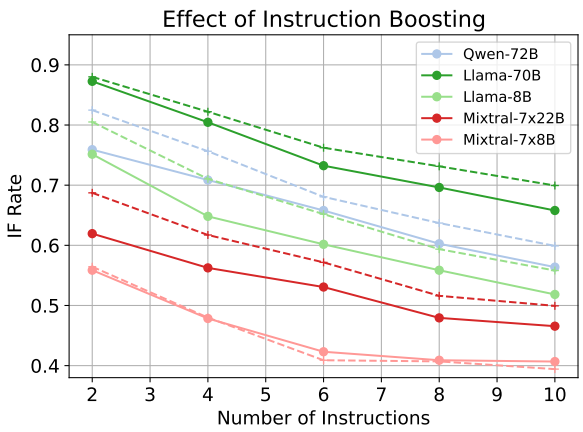

Figure 3: Initial IF rate (solid lines) and Best-of-N boosting performance (dashed lines).

Note that across all models the IF rate drops as the number of instructions is increased, confirming previously observed instruction following degradation at scale. We will return to examine the factors that contribute to these degradation trends in Section 4.

3.1 Boosting Strategies

Instruction boosting is a test-time post-generation improvement strategy. A boosting strategy takes as input a query, a set of instructions and a generated response and produces as output a new response that has been revised to maximize adherence to the instructions. We devised the following boosting strategies (see Appendix for the corresponding prompts).

Detect+Repair: Detect+Repair proceeds in two steps. First, an LLM-as-a-judge detector (judge detector) is used to determine which instructions have not been followed in the input response. In the second repair step the response is rewritten to repair all detected instruction violations.

Best-of-N: Best-of-N samples N rewritten responses that are to follow all instructions not already adhered to in the initial response using temperature sampling. Best-of-N does not require an initial detection step. Instead, the judge detector is used as a reward model to assign an instruction following reward to each rewritten response. The reward is the IF rate detected by the judge detector. In a final selection step, the response with the highest reward is chosen as the repaired response. In all experiments we used $N = 5$ as we observed

[1](Meta AI, 2024a): Meta-Llama-3.3-70B-Instruct
[2](Meta AI, 2024b): Meta-Llama-3.1-8B-Instruct
[3](Alibaba Cloud, 2024): Qwen2.5-72B-Instruct
[4](Mistral AI, 2023): Mixtral-8x7B-Instruct-v0.1
[5](Mistral AI, 2024b): Mixtral-8x22B-Instruct-v0.1

diminishing returns at higher sampling rates.

Best-of-N Oracle: To understand the potential IF rates models can achieve in their rewrites, we include a variant of Best-of-N that uses an oracle reward model to assign the actual instruction following rate to a given response. We used the deterministic IFEval instruction verifiers as the oracle.

Map Reduce: Map Reduce proceeds in three phases. First, the judge detector is used to detect violated instructions in the initial response. The Map phase creates separate rewrite tasks for each detected instruction violation. The final Reduce phase merges the independently generated rewritten responses into one final repaired response.

Fig. 4 shows the achieved IF rates for the four strategies for Llama-70B, the best performing model from Fig. 3. The IF rate achieved in the initial responses, prior to boosting is shown as the baseline. All boosting strategies lead to IF rate improvements over the baseline. Even at two instructions, boosting leads to small improvements and the benefits generally increase with the number of instructions.

Among the non-oracle strategies, Best-of-N consistently provides the largest boost, up to 4 percentage points at ten instructions, increasing the IF rate to 0.70 from 0.66. Best-of-N Oracle shows the potential IF rate achievable through rewrite sampling. Even at two instructions, the model is capable of generating rewritten responses with an IF rate of 0.89, a 2 percentage point increase. The boost grows as instructions are increased to ten, when the IF rate reaches 0.75, an 8.5 percentage point increase. Although not shown for space constraints, the relative boosting trends across the fours strategies are similar across all models from Fig.3.

The gap between Best-of-N and Best-of-N Oracle is a result of inaccuracies in the judge detector that is used as the reward model. When using Llama-70B as the reward model, the detection accuracy is 73%. Thus, closing this gap may be achieved by replacing the judge detector with manually coded or LLM generated deterministic verifiers.

Task Adherence: Since the instructions are mostly orthogonal to the query in each sample, it is possible to satisfy them while completely ignoring the primary task, namely answering the query. As a quality check, we used Llama-70B as a task-adherence judge on the initial model response and on the response after instruction boosting. This task-adherence judge was instructed to determine if the given response was related to the query (see the Appendix for the judge prompt that was used). In the case that the initial response fails this check, the data point is not included in the results. If the initial response passed but after instruction boosting the response failed, this is considered an additional failure mode. Table 1 shows that the largest number of task adherence failures in the initial response generation was incurred for 8 instructions with 22 out of 538 (4%) failures. Boosting caused at most 7 (1.3%) additional task adherence failures in the 10 instruction experiment.

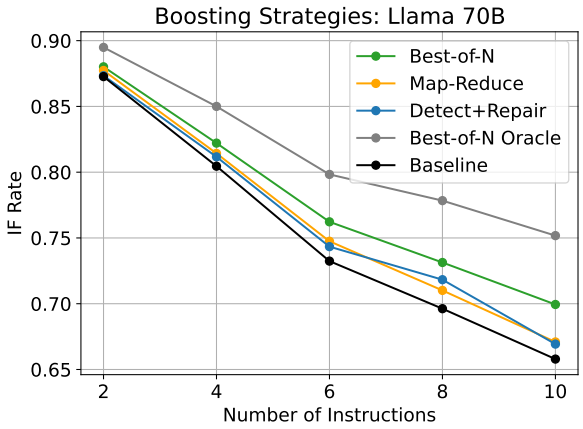

Figure 4: Instruction following rate achieved by Llama 70b for different boosting strategies.

Num Instructions	2	4	6	8	10
Initial Failures	0	11	8	22	20
Add'l after boosting	0	0	6	3	7

Table 1: Task Adherence Failures (Llama-70B) out of 538 samples.

3.2 Cost Analysis

In addition to the varying benefits among the boosting strategies we considered, they also incur different cost-benefit tradeoffs. To illustrate these tradeoffs Fig. 5 plots the achieved IF rate against cost measured as completion tokens in Flops. Compared to the lowest cost Detect+Repair strategy, Best-of-N trades additional compute for an additional IF rate boost. Map Reduce is less cost effective, requiring significantly more compute for a small IF rate increase.

We also explored the following boosting variations.

Detect+Repair Expl: As a variant of De-

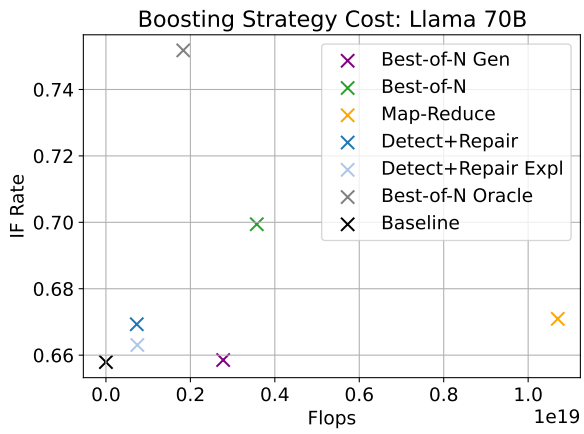

Figure 5: Costs (completion tokens in Flops) and IF rate of each strategy for 10 instructions achieved by Llama-70B.

tect+Repair, we added explanations of instruction violations to the rewrite prompt. We expected the additional explanation hints to help the model in the rewrite task. Surprisingly, adding explanations didn't help and may have only provided a distraction to the model since they actually led to a small erosion of the IF rate improvements as shown in Fig. 5.

Best-of-N Gen: Instead of sampling N response rewrites, Best-of-N Gen samples N initial response generations to the query. Both Best-of-N and Best-of-N Gen use the judge detector as a reward model. As shown in the Fig. 5, compared to rewrite sampling, sampling the initial responses incurs slightly lower cost but was not able to match the IF rate improvements of Best-of-N confirming our hypothesis that repairing a non-compliant draft response is generally easier than writing a compliant response from scratch.

4 Instruction Scaling and Conflict Analysis

This section takes a closer look at the drivers for the observed instruction scaling trends that show decreasing IF rates with increasing numbers of instructions.

4.1 Soft Conflicts

In Section 2, we described how we applied predefined constraints to avoid contradictory instructions. For example, an instruction that requires the keyword *'confidential'* to appear in the response would contradict one that prohibits the same word. We refer to a pair of contradictory instructions as a *hard conflict*. Even after applying these pre-defined constraints and ruling out hard conflicts, there may still be instructions that are difficult for a model to follow simultaneously. For example, instructing a model to include at least 300 words in a response and also instructing it to not repeat any words may be difficult. We carry out a self-play approach to empirically identify such *soft conflicts*.

Num instr.	2	4	6	8	10
Conflict score	0.24	0.67	1.17	1.59	2.03
Correlation	-0.79	-0.63	-0.46	-0.42	-0.37

Table 2: Estimated average conflict scores (after boosting) for Best-of-N boosting with Llama-70B.

To quantify the degree of soft conflict in a sample, we compute a *conflict score* between every pair of instructions assigned to that sample. The conflict score of a pair of instructions indicates the difficulty of following both instructions simultaneously. Conflict scores are computed by sampling multiple responses from a model for each instruction pair. Each response is checked to determine which instructions were followed: either both instructions were followed, or at least one was not followed. The latter case indicates possible soft conflicts. This flow is depicted in Figure 6.

More precisely, we start from the original dataset D, outlined in Section 2, with N samples, where each sample consists of a query and k instructions. Next we construct a *pairwise* dataset $\hat{D}$ as follows: for each sample in D we create $k(k-1)$ samples with the same query and every subset of two instructions from the original k instructions. The model generates r responses for each sample in $\hat{D}$ to produce the response set $\hat{R}$. The conflict count c_{ij} between instructions i and j is the number of responses in $\hat{R}$ where at least one of the two instructions i or j were not followed.

We can now use the pair-wise conflict counts to compute a conflict score for each sample $s \in D$. Let $p(s)$ be the set of instructions associated with sample s. The conflict score c_s of sample s is the normalized sum of the conflict counts for each pair of instructions in s:

$$c_s = \frac{\sum_{(i,j)\in p(s)\times p(s), i\neq j} c_{ij}}{|p(s)|}. \quad (1)$$

Conflict score at scale: We expect that samples with higher conflict scores include instructions that are more difficult for a model to simultaneously follow and boost. We see this in Table 2, where the

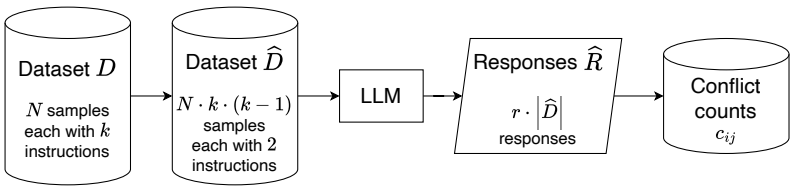

Figure 6: Steps to empirically compute an estimated conflict score between a pair of instructions.

average conflict scores increase with the number of instructions in the dataset. This suggests that part of the reason that instruction following compliance rate decreases with the number of instructions, as observed in Section 3, is that there are inherent conflicts when more instructions are added. Recall that the conflict scores are computed pair-wise so the conflict scores themselves are not susceptible to instruction scaling effects.

Correlation of conflict score: Table 2 also lists the Pearson correlation between the IF rate and the conflict scores. The correlation decreases with increasing instructions; in the case with 10 instructions, the correlation is about -0.37.

We posit two reasons for the decreasing correlation. First, our conflict scores are only based on pair-wise instruction conflicts. There may be more complex conflicts that only arise in the interplay of more than two instructions. For example, consider a set of three instructions that place limits on the total number of words, the number of sentences, and the number of words per sentence in the response. Following any pair of these instructions is easier than following all three.

Furthermore, while our conflict analysis shows that soft conflicts are an important contributor to degrading IF rates with increasing numbers of instructions, their presence may not be the sole driver for this trend. Additional factors may play a role that cause IF degradations even in the absence of any soft conflicts among instructions. Further investigating and quantifying the contributions of instruction conflicts and other causes is an avenue for future work.

Segmentation by IF rate: To take a closer look into the contribution of conflict scores to degrading IF rates, we examine the distribution of conflict scores across IF rate segments. Figure 7 plots the average conflict score for samples bucketed by the IF rate measured for the dataset with ten instructions. We see that "harder" samples, i.e., those with a lower IF rate, do indeed have a higher conflict score. This reinforces our assertion that soft conflicts among instructions contribute to worsening IF rates when there are many instructions.

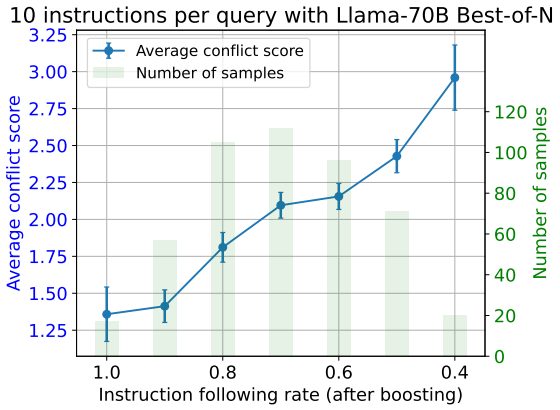

Figure 7: Average conflict scores for 10 instruction dataset, bucketed by the IF rate after boosting.

Note that the error bars are the standard error of the mean, which is likely an underestimate of the true uncertainty due to correlations between instruction pairs appearing in multiple samples.

Comparison without constraints: Recall from Section 2 that we apply pre-defined parameter sampling constraints to avoid including contradicting instructions (i.e, instruction pairs with hard conflicts) in the dataset. We repeated the above analysis on a dataset where we did not apply these pre-defined constraints. As expected we observed a higher average conflict score: with 10 instructions the average conflict score was 18% higher.

4.2 Lost-in-the-Middle

Prior work (Liu et al., 2023) looked into the "lost-in-the-middle" effect and observed that performance is often highest when relevant information occurs at the beginning or end of the LLM input context, and significantly degrades when models must access relevant information in the middle.

We investigated whether the lost-in-the-middle effect could play a role in the lowering IF rates we observed at rising numbers of instructions. Large numbers of instructions also have larger numbers

of "middle" instructions. To investigate, we broke down the IF rate by instruction position to compute positional IF rates as the n-th position IF rate. However, we found no consistent relationship between IF rates and instruction position across models. Middle instructions generally did not have lower IF rates than first or last instructions. Thus, the larger number of "middle" instructions in a list of 10 instructions does not appear to be a driver for the IF rate degradation we observed.

5 Related Work

Instruction following is critical to LLMs, enabling them to align with human intent. Related work on evaluating a model's instruction following ability includes IFEval (Zhou et al., 2023) which focuses on verifiable instructions that can be checked deterministically. IFEval includes data samples with one to three instructions per sample. ScaledIF extends IFEval by scaling instruction volume up to ten instructions per data sample. Multi-IF (He et al.) expands on IFEval to include multi-turn conversations in multiple languages. MultiTurnInstruct (Han, 2025), ComplexBench (Wen et al., 2024) and FollowBench (Jiang et al., 2024) evaluate how LLMs handle complex instructions with different types of constraints and their compositions. InFoBench (Qin et al., 2024) introduces a new metric (DRFR) that breaks down complex instructions into simpler criteria for a more detailed analysis of a model's compliance with each part of the instruction. RefuteBench (Yan et al., 2024) focuses on evaluating whether a model can modify its response to comply with human feedback in the form of refuting instructions in a conversational context. Verbalizer manipulation (Li et al., 2024) introduces an instruction following evaluation protocol for classification tasks. IFScale (Jaroslawicz et al., 2025) is a benchmark of 500 keyword-inclusion instructions for a business report writing task and LIFBench'(Wu et al., 2025) focuses on evaluating instruction following in long context scenarios.

Beyond evaluation, prior research has explored test-time interventions to improve reasoning and overall response quality. An approach similar to Detect+Repair boosting is self-correction (Madaan et al., 2023; Huang et al., 2024) where a model is prompted to evaluate its own initial output and refine it. Self-correction primarily focuses on improving reasoning quality, whereas instruction boosting specifically targets a model's instruction following ability.

Although not specifically aimed at instruction following, Chain-of-Thought (CoT) prompting (Wei et al., 2023) can improve model performance on complex tasks by breaking down a difficult problem into smaller, more manageable steps. Self-Consistency (Wang et al., 2023) builds on CoT prompting by sampling multiple model responses and selecting the most frequent or "consistent" response. Similar to Best-of-N boosting, self-consistency leverages a model's inherent capabilities to find a robust solution through compute scaling (Snell et al., 2024). However, in contrast to self-consistency which samples response generation with the goal to improve CoT reasoning, Best-of-N boosting samples response rewrites and is targeted at improving IF rates.

(Wu et al., 2024) propose a training method for instruction following with thought generation and (Hou et al., 2025) propose a dynamic pruning approach to improve instruction following. In contrast, instruction boosting aims to improve instruction following as a post-processing method.

6 Conclusion

In this work, we've demonstrated that prompt-based instruction following in LLMs is a challenging problem, exacerbated by the instruction scaling effect. We introduced Instruction Boosting, a test-time strategy to enhance instruction adherence by refining and correcting initial model responses. Our work also contributes a dataset to systematically study instruction following and the effects of boosting at increasing numbers of instructions. Our empirical results on the ScaledIF dataset confirm that instruction boosting consistently improves instruction following, with gains of up to 7 and 4 percentage points for 2 and 10 instructions, respectively. We further contributed the concept of soft conflicts and a quantitative conflict score to explain observed instruction scaling trends and show how conflict scores can be serve as a diagnostic tool for developers. By providing clear feedback to anticipate instruction following difficulty, our approach offers both a powerful method for improving model reliability and a valuable feedback mechanism for more effective prompt engineering and control over model behavior in applications.

7 Limitations

This work focused on syntactic instructions which can be deterministically checked. More ambiguous instructions that may require a human or LLM judge to validate are the subject of future work. In addition, the LLMs were required to follow all instructions provided in their prompt. Real-world applications can include irrelevant distractor instructions which are outdated or dependent on an unfulfilled condition.

References

Alibaba Cloud. 2024. Qwen2.5-72b-instruct. https://huggingface.co/Qwen/Qwen2.5-72B-Instruct. Accessed: 08/15/2026.

Chi Han. 2025. Can language models follow multiple turns of entangled instructions? *arXiv preprint arXiv:2503.13222*.

Yun He, Di Jin, Chaoqi Wang, Chloe Bi, Karishma Mandyam, Hejia Zhang, Chen Zhu, Ning Li, Tengyu Xu, Hongjiang Lv, and 1 others. Multi-if: Benchmarking llms on multi-turn and multilingual instructions following, 2024. *URL https://arxiv.org/abs/2410.15553*.

Bairu Hou, Qibin Chen, Jianyu Wang, Guoli Yin, Chong Wang, Nan Du, Ruoming Pang, Shiyu Chang, and Tao Lei. 2025. Instruction-following pruning for large language models. In *ICML 2025*, arXiv:2501.02086.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2024. Large language models cannot self-correct reasoning yet. In *ICLR 2024*, arXiv:2310.01798.

IBM IBM Granite. 2024. Granite 3.0 language models.

Daniel Jaroslawicz, Brendan Whiting, Parth Shah, and Karime Maamari. 2025. How many instructions can llms follow at once? *arXiv:2507.11538*.

Yuxin Jiang, Yufei Wang, Xingshan Zeng, Wanjun Zhong, Liangyou Li, Fei Mi, Lifeng Shang, Xin Jiang, Qun Liu, and Wei Wang. 2024. Followbench: A multi-level fine-grained constraints following benchmark for large language models. *arXiv preprint arXiv:2310.20410*.

Shiyang Li, Jun Yan, Hai Wang, Zheng Tang, Xiang Ren, Vijay Srinivasan, and Hongxia Jin. 2024. Instruction-following evaluation through verbalizer manipulation. In *NAACL 2024*, arXiv:2307.10558.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. *arXiv:2303.17651*.

Meta AI. 2024a. Meta-llama-3.1-70b-instruct. https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct. Accessed: 08/15/2026.

Meta AI. 2024b. Meta-llama-3.1-8b-instruct. https://huggingface.co/meta-llama/Meta-Llama-3.1-8B-Instruct. Accessed: 08/15/2026.

Mistral AI. 2023. Mixtral-8x7b-instruct-v0.1. https://huggingface.co/mistralai/Mixtral-8x7B-Instruct-v0.1. Accessed: 08/15/2026.

Mistral AI. 2024a. Mistral Large. Available at https://docs.api.nvidia.com/nim/reference/mistralai-mistral-large (or a more specific resource where you accessed the model).

Mistral AI. 2024b. Mixtral-8x22b-instruct-v0.1. https://huggingface.co/mistralai/Mixtral-8x22B-Instruct-v0.1. Accessed: 08/15/2026.

Yiwei Qin, Kaiqiang Song, Yebowen Hu, Wenlin Yao, Sangwoo Cho, Xiaoyang Wang, Xuansheng Wu, Fei Liu, Pengfei Liu, and Dong Yu. 2024. Infobench: Evaluating instruction following ability in large language models. *arXiv:2401.03601*.

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

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. In *ICLR 2023*, arXiv:2203.11171.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *arXiv:2201.11903*.

Bosi Wen, Pei Ke, Xiaotao Gu, Lindong Wu, Hao Huang, Jinfeng Zhou, Wenchuang Li, Binxin Hu, Wendy Gao, Jiaxin Xu, Yiming Liu, Jie Tang, Hongning Wang, and Minlie Huang. 2024. Benchmarking complex instruction-following with multiple constraints composition. In *NeurIPS 2024*, arXiv:2407.03978.

Tianhao Wu, Janice Lan, Weizhe Yuan, Jiantao Jiao, Jason Weston, and Sainbayar Sukhbaatar. 2024. Thinking llms: General instruction following with thought generation. *arXiv:2410.10630*.

Xiaodong Wu, Minhao Wang, Yichen Liu, Xiaoming Shi, He Yan, Xiangju Lu, Junmin Zhu, and Wei Zhang. 2025. Lifbench: Evaluating the instruction following performance and stability of large language models in long-context scenarios. *arXiv:2411.07037*.

Jianhao Yan, Yun Luo, and Yue Zhang. 2024. Refutebench: Evaluating refuting instruction-following for large language models. In *ACL 2024*, arXiv:2402.13463.

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

A SCALEDIF Construction

The following 26 instruction classes were used in the construction of the SCALEDIF dataset.

length_constraints	sentence_length*
	number_sentences
	number_words
	number_paragraphs
	nth_paragraph_first_word
detectable_format	number_bullet_lists
	constrained_response
	number_highlighted_sections
	title
	multiple_sections
	json_format
	yaml_format*
detectable_content	number_placeholders
	postscript
punctuation	no_comma
keywords	forbidden_words
	frequency
	letter_frequency
	existence
combination	repeat_prompt
	multiple_responses
change_case	capital_word_frequency
	lowercase_sentences
startend	quotation
	start_checker*
	end_checker

Table 3: SCALEDIF Instructions, * indicates instructions not present in IFEval.

All but three (length_constraints:sentence_length, detectable_format:yaml_format, startend:start_checker) of these instructions were part of the original IFEval dataset. In addition, three of the original IFEval instructions (response_language, english_uppercase, english_lowercase) were removed because they created too many conflicts with other instructions to allow long, conflict-free sets.

A.1 Constraints

In order to ensure that the sets of instructions constructed for SCALEDIF are mutually satisfiable, constraints had to be imposed on the sampled parameters. For example suppose that length_constraints:number_paragraphs is sampled with parameters num_paragraphs = 3, and then length_constraints:number_words is sampled. Before sampling the parameters for length_constraints:number_words, all constraints related to the previously sampled instructions will be computed. The length_constraints:number_paragraphs instruction will produce the constraint num_words $>= 10*$ num_paragraphs $= 30$. This constraint is chosen to ensure that the response is allowed to have enough words to form the necessary number of paragraphs. As another example, the keywords:forbidden_words and keywords:existence instructions each have a list of keywords as their parameter. If both of these instructions are sampled, then the second one chosen is required to exclude the keywords already taken by the first one.

There is additionally one three-way constraint imposed between length_constraints:sentence_length, length_constraints:number_sentences, and length_constraints:number_words.

$$\text{N\_words} >= (\text{N\_sentences}+3)*(\text{sentence\_length}+3) \quad (2)$$

A.2 Example

Table 4 shows one representative set of 10 instructions, which were assigned in ScaledIF to the query: "Can you help me write a letter to a company that is refusing to refund a product I bought online?"

B Boosting Strategy Details

Below we include the prompt templates used in the boosting strategies.

B.1 Detect+Repair

Detect Prompt:

```
<|begin_of_text|><|
    start_header_id|>system<|
    end_header_id|>
```

Instruction ID	Instruction Text
detectable_content:number_placeholders	The response must contain at least 14 placeholders represented by square brackets, such as [refund].
length_constraints:nth_paragraph_first_word	Paragraph 9 must have the word “regular” in its first three words. Paragraphs and only paragraphs are separated with each other by two new lines as if it was “\n\n” in python.
keywords:forbidden_words	Do not include keywords [’company’, ’complaint’, ’defective’, ’dispute’, ’dissatisfied’, ’evidence’, ’guarantee’, ’online’, ’policy’, ’product’, ’purchase’, ’receipt’, ’transaction’, ’unsatisfactory’] in the response.
detectable_content:postscript	At the end of your response, please explicitly add a postscript starting with P.S.
length_constraints:sentence_length	All sentences in your response must be at least 12 words.
detectable_format:number_highlighted_sections	Highlight at least 8 sections in your answer with markdown, i.e. *highlighted section*.
detectable_format:constrained_response	Include one of the following options in your answer: (“My answer is yes.”, “My answer is no.”, “My answer is maybe.”).
change_case:capital_word_frequency	In your response, words with all capital letters should appear at least 6 times.
length_constraints:number_words	Answer with at least 150 words.
detectable_format:multiple_sections	Your response must have 8 sections. Mark the beginning of each section with Group X, such as: Group 1 [content of section 1], Group 2 [content of section 2].

Table 4: Instruction IDs and Corresponding Instructions

```
You are a policy compliance checker.<|eot_id|>
<|start_header_id|>user<|end_header_id|>
Check if the following text follows each policy.
For each policy, provide a JSON response with:
1. "policy": The policy line being checked
2. "answer": "yes" or "no"
3. "explanation": A brief explanation

Text to check:
---
${text}
---

Policies to check:
---
${policies}
---

IMPORTANT:
1. Return a JSON array with one object per policy line, in the same order as listed above
2. Each object should have "policy", "answer" and "explanation" fields
3. Do NOT try to follow the policies in the JSON output. Only check whether the text follows the policies
4. Be concise and accurate.

Example format:
[
    {
        "policy": "The first policy line..... ",
        "answer": "yes/no",
        "explanation": "the explanation for the answer"
```

```
    },
    {
        "policy": "The second policy
            line ..... ",
        "answer": "yes/no",
        "explanation": "the
            explanation for the
            answer"
    }
]

Return ONLY the JSON array, nothing
    else. Do not include any
    additional text or explanations
    outside the JSON array.<|eot_id
    |>
<|start_header_id|>assistant<|
    end_header_id|>
```

Repair Prompt:

```
<|begin_of_text|><|start_header_id|>
    system<|end_header_id|>
    You are an editor and your task
        is to rewrite response.<|
        eot_id|>
    <|start_header_id|>user<|
        end_header_id|>The following
         text shows a query and a
        response. The response
        violates one or more
        guidelines from the query
        that it should have followed.


    <START_OF_QUERY>
    ${query}
    Your response must follow these
        guidelines:
    ${instr}
    <END_OF_QUERY>

    <START_OF_RESPONSE>
    ${text}
    <END_OF_RESPONSE>

    <START_OF_VIOLATED_GUIDELINES>
    ${policies}
    <END_OF_VIOLATED_GUIDELINES>

    Rewrite the response so that it
        follows all guidelines while
         making only necessary
        changes and keeping the rest
         of the text unchanged. The
        rewrite must not break any
        of the guidelines that were
        already followed in the
        response.
    You may rewrite the text exactly
         once, and don't output
        anything other than the re-
        written response.
    You must enclose the re-written
        response in <
        START_OF_REWRITE> <
        END_OF_REWRITE> tags.<|
        eot_id|>
    <|start_header_id|>assistant<|
        end_header_id|>
```

B.2 Best-of-N

```
    <|begin_of_text|><|
        start_header_id|>system<|
        end_header_id|>
    You are an editor and your task
        is to rewrite a response to
        ensure compliance with
        guidelines.<|eot_id|>
    <|start_header_id|>user<|
        end_header_id|>The following
         text shows a query, a
        response and a set of
        guidelines. The response may
         violate one or more
        guidelines that it should
        have followed.

    <START_OF_QUERY>
    ${query}
    <END_OF_QUERY>

    <START_OF_RESPONSE>
    ${text}
    <END_OF_RESPONSE>

    <START_OF_GUIDELINES>
    ${instr}
    <END_OF_GUIDELINES>

    Rewrite the response so that it
        follows all guidelines while
         making only necessary
        changes and keeping the rest
         of the text unchanged. The
        rewrite must not break any
```

```
        of the guidelines that were
        already followed in the
        response.
    You may rewrite the text exactly
         once, and don't output
        anything other than the re-
        written response.
    You must enclose the re-written
        response in <
        START_OF_REWRITE> <
        END_OF_REWRITE> tags.<|
        eot_id|>
    <|start_header_id|>assistant<|
        end_header_id|>
```

B.3 MapReduce

```
    <|begin_of_text|><|
        start_header_id|>system<|
        end_header_id|>
You are an editor and your task is
    to generate a response according
     to a user query and to ensure
    compliance with a set of
    guidelines.<|eot_id|>

<|start_header_id|>user<|
    end_header_id|>The following
    text shows the user query, a set
     of per-guideline responses, and
     the set of guidelines. Each per
    -guideline response is an
    example of a response that
    complies with the associated
    guideline.

<START_OF_QUERY>
${query}
<END_OF_QUERY>
${per_guideline_responses}
<START_OF_GUIDELINES>
${instr}
<END_OF_GUIDELINES>

Generate a response so that it
    follows all the guidelines. Use
    the per-guideline responses to
    help generate the final response
     that complies with all the
    guidelines.
Don't output anything other than the
     re-written response.
You must enclose the re-written
    response in <START_OF_REWRITE> <
    END_OF_REWRITE> tags.<|eot_id|>
<|start_header_id|>assistant<|
    end_header_id|>
```

B.4 Best-of-N Gen

```
    <|begin_of_text|><|
        start_header_id|>system<|
        end_header_id|>
    You are a helpful assistant and
        your task is to respond to
        queries. Your response must
        follow a set of guidelines
        .<|eot_id|>
    <|start_header_id|>user<|
        end_header_id|>Here is a
        query with a set of
        guidelines that must be
        followed.

    <START_OF_QUERY>
    ${query}
    Your response must follow these
        guidelines:
    ${instr}
    <END_OF_QUERY>

    Write a response that follows
        all guidelines and don't
        output anything other than
        the response.
    You must enclose the response in
         <START_OF_RESPONSE> <
        END_OF_RESPONSE> tags.<|
        eot_id|>
    <|start_header_id|>assistant<|
        end_header_id|>
```

C Task Adherence Judge

```
    <|begin_of_text|><|
        start_header_id|>system<|
        end_header_id|>
    You are a helpful assistant
        whose task is to respond to
        user queries.
    Make sure that your response
        follows these instructions:

    {instruction_list}<|eot_id|>
    <|start_header_id|>user<|
        end_header_id|>
```

```
    Question:
    {prompt_request}<|eot_id|>
    <|start_header_id|>assistant<|
        end_header_id|>
```