What Prompts Don't Say: Understanding and Managing Underspecification in LLM Prompts

Chenyang Yang Yike Shi Qianou Ma Michael Xieyang Liu*
Christian Kästner Tongshuang Wu
Carnegie Mellon University
https://github.com/malusamayo/underspec-analysis

Abstract

Building LLM-powered software requires developers to communicate their requirements through natural language, but developer prompts are frequently underspecified, failing to fully capture many user-important requirements. In this paper, we present an in-depth analysis of prompt underspecification, showing that while LLMs can often (41.1%) guess unspecified requirements by default, such behavior is less robust: Underspecified prompts are 2x more likely to regress over model or prompt changes, sometimes with accuracy drops by more than 20%. We then demonstrate that simply adding more requirements to a prompt does not reliably improve performance, due to LLMs' limited instruction-following capabilities and competing constraints, and standard prompt optimizers do not offer much help. To address this, we introduce novel requirements-aware prompt optimization mechanisms that can improve performance by 4.8% on average over baselines that naively specify everything in the prompt. Beyond prompt optimization, we envision that effectively managing prompt underspecification requires a broader process, including proactive requirements discovery, evaluation, and monitoring.

1 Introduction

As large language models (LLMs) are improving in capabilities and instruction following [30], they are increasingly integrated into business applications [e.g., 3, 7] through customized instruction *prompts* that can span thousands of words [e.g., 1, 2]. Conveying nuanced intentions to LLMs, however, is inherently difficult. For end users interacting directly with a model (e.g., ChatGPT), this issue is less significant, as their prompts are typically one-off and considered successful as long as they yield one satisfactory response. For *developers* of LLM-powered applications, the problem is much more serious, as their prompts need to generalize to many different user inputs.

As an example (Figure 1), think about a developer working on an LLM-powered trip advisor application: The developer builds a prompt that specifies LLMs' task ①, tone ②, and various behaviors, including avoiding transaction handling ③ and clarify ambiguity ④. This prompt is detailed yet still *underspecified* in many aspects: For example, it does not specify whether LLMs should alert users of potential weather conditions, proactively ask follow-up questions, or remind visa (entry) requirements for suggested destinations. If the LLM ends up not satisfying these requirements, it can cause frustrations and failures, such as users booking travel activities during bad weather, receiving vague recommendations, or facing denied entry due to visa issues, ultimately undermining trust in the LLM-powered applications. Indeed, failing to mention prerequisites of suggested activities has already led to a chaotic user experience with an existing LLM-powered trip advisor [6].

*Now at Google Deepmind.

Preprint. Under review.

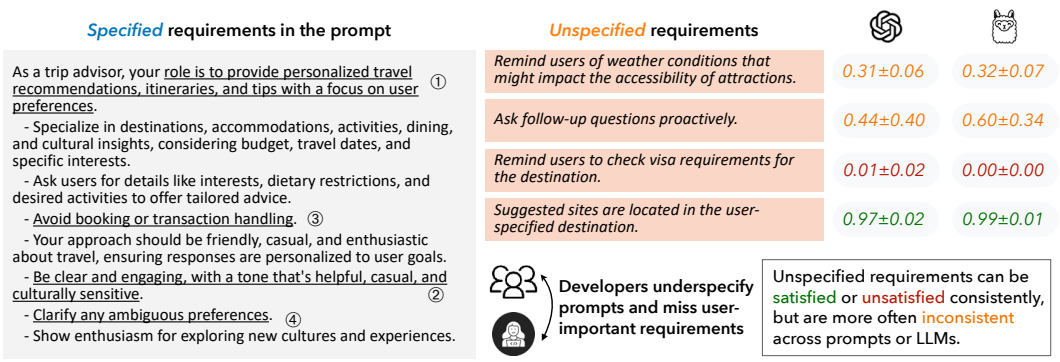

Figure 1: Developers often underspecify their prompts and miss user-important requirements. These unspecified requirements might be guessed by LLMs by default, but are more often inconsistent across prompts or LLMs. Such underspecification can cause frustration and failures for the end users.

While it is possible that developers simply do not care enough to specify these behaviors, it is equally – if not more – likely that current engineering and evaluation practices make it difficult for developers to identify such underspecification until they have already led to issues in deployment (Section 2). Even if developers do not believe these behaviors must be explicitly defined, they may expect models to act consistently along these dimensions to support more stable user mental models of LLM applications, which is, however, not the case for many unspecified requirements (Figure 1).

In this work, we characterize (Section 2) and empirically analyze (Section 3) the problem of prompt underspecification. To systematically study the effects of underspecification, we curated a set of diverse requirements across 3 representative tasks, constructed a series of prompts that specify different subsets of the requirements, and evaluated each requirement's satisfaction rate with human-validated LLM-as-a-judge [48] on all prompts. In total, we collected 8.4k data points of LLM+Prompts' aggregated behaviors on diverse requirements. Our analysis demonstrated that, *while LLMs can indeed often (41.1%) fill in the underspecification gap, their behaviors are rather inconsistent:* One version of an LLM may excel in fulfilling an unspecified requirement, but the next version can unexpectedly degrade by more than 20%. This will be a problem for continuously developing, deploying, and maintaining LLM-powered applications reliably.

We then introduce *requirements-aware prompt optimization* as a solution strategy to deliberately communicate important requirements to the model, while leaving those already implicitly fulfilled unspecified. We show that such strategies overcome issues with existing approaches: The obvious strategy of simply *specifying all requirements* in the prompt does not work, due to LLMs' limited instruction-following capabilities – their performance can drop by 19% as we specify more requirements (Section 3.5), and requirement-agnostic prompt optimization only provides limited help since they have no requirement-specific feedback. We propose and evaluate two *requirement-aware* prompt optimizers (Section 4): One to optimize *how* to specify requirements, and the other to explicitly optimize *what* requirements to specify. We demonstrate that both strategies work well (+4.8% accuracy), and the latter can produce shorter prompts (-43% tokens) that are easier to follow for the model.

Finally, we discuss what it takes to properly *manage* prompt underspecification when building LLM-powered applications in practice beyond prompt optimization (Section 5): This includes proactively discovering important requirements, building reliable requirement evaluators, as well as continuously evaluating and monitoring (un-)specified requirements. We highlight the research opportunities here to support the entire process of managing prompt (under-)specification.

In summary, our work makes the following contribution: **(1)** Characterization of the underspecification problem in LLM prompts (Section 2), **(2)** an empirical analysis of LLM+Prompt behaviors when underspecified (Section 3), **(3)** mitigation mechanisms with *requirements-aware* prompt optimizers (Section 4), and **(4)** a discussion of our vision for managing prompt underspecification (Section 5).

2 Underspecification is Amplified in LLM Prompts

While underspecification is a known challenge in traditional software engineering and machine learning pipelines [17, 11], it is significantly exacerbated in the context of LLM prompting. In this

Table 1: Compared to traditional ML models and software, LLM prompts are more prone to underspecification, exhibit greater instability, and evolve more frequently. Prompt engineering practices and the dynamic LLM+prompt interplay both amplify these issues.

Aspect	LLM Prompt	Traditional ML Models	Traditional Software
Specification Method	Natural language prompts (instructions, examples)	Training data and model architecture and pipeline	Usually natural language and sometimes formal specifications
Engineering Practices	Ad-hoc, trial-and-error prompt iteration [22, 43]	More structured experimentation pipelines [16]	Systematic requirements engineering and design processes [40]
Artifact Evolution	Frequent changes with little version control [39, 22]	Periodical evolution with some version control [16]	Evolution often intentionally tracked and limited [25, 8]
Behavioral Stability	Highly sensitive to prompt phrasing and LLM version [9, 24]	Moderate variability across training runs and datasets [11]	Mostly deterministic and version-controlled
Symptoms of Under-specification	Inconsistent and unexpected LLM behaviors (Section 3)	Generalization failures, bias due to data underspecification [11]	Software that does not meet customer needs; incorrect behavior in edge cases [36, 21]

section, we articulate unique characteristics in LLM prompts that make underspecification more challenging: Unlike traditional systems, prompt engineering is informal, fast-changing, and rarely guided by systematic engineering practices [22, 43]. Moreover, LLM behaviors are highly sensitive to prompt phrasing and LLM versions, which further amplifies the effects of underspecification. A summary of comparison is in Table 1.

Prompt engineering practices exacerbate underspecification. Prompts are developed with the expectation that not everything needs to be specified – ideally, LLMs should behave like a human and fill in the gaps with commonsense. This expectation encourages a *"minimal-specification"* prompt engineering practice: Developers begin with an initial prompt, observe violations of expected behavior, and iteratively revise through adding more instructions (i.e., specifications) [43, 22]. This trial-and-error process lacks the rigor of traditional requirements engineering [40], and exposes only a narrow slice of possible behaviors. This leaves plenty of room to miss important behaviors as developers look only at a few examples and do not necessarily know what they are looking for (Section 3.4).

Prompts and LLMs evolve much more frequently together. Prompts are not static artifacts – they are routinely modified to add new features, address failures, or adapt to updated LLMs [39]. Simultaneously, LLMs themselves undergo frequent version changes, often without clear change logs, and sometimes silently without developers' control [24, 10]. This dual axis of prompt and model changes introduces behavioral drift in addition to traditional requirements changes. In contrast, traditional software change is often carefully managed [25, 8] and most ML pipelines evolve more slowly.

LLM behaviors are highly unstable and sensitive to (under-)specification. LLM behaviors are known to be unstable and sensitive to prompt wordings [9]. Similarly, given the same prompt, different LLMs exhibit different behaviors [13], even just across small version updates [10]. As we will show later (Section 3), LLM behaviors are especially unstable in underspecified settings, where LLM behaviors depend on implicit assumptions that are not guaranteed to transfer across changes.

3 How do LLM+Prompts Behave when Underspecified?

While LLM+Prompts are generally known to be unstable, existing work mostly studies their stability on *specified* requirements in the prompts. To quantitatively measure LLM+Prompts' behaviors on *unspecified* requirements, we design an experiment as follows: We collect a set of 60 plausible requirements across 3 tasks and create validated, automated validators for each requirement. We then create prompts with subsets of the requirements and measure how well the model's outputs meet the (un-)specified requirements using their validators.

Our analysis starts with showing that LLMs can often guess unspecified requirements, but these behaviors are inconsistent (Section 3.2) and more likely to degrade with model updates (Section 3.3). We next analyze why identifying unspecified requirements is fundamentally hard with current

practices, especially when requirements are conditional or rarely violated (Section 3.4). Finally, we show that an obvious solution of specifying all requirements in a single prompt actually hurts performance, due to LLMs' limited ability to follow long, complex instructions (Section 3.5).

3.1 Experiment Setups

Tasks and data. We selected three tasks based on Anthropic's report measuring AI usage patterns [14], covering diverse tasks that can be integrated in commercial applications, from software engineering (`code-explain`), travel industry (`trip-advisory`), to E-commerce business (`product-gen`).

We re-purposed three existing datasets to run the LLM+Prompts on: Commitpackft [27] for code explanation, a subset of UltraChat [12] for trip advisory, and Amazon ESCI [32] for product description generation. From each dataset, we sampled 200 examples, and split them into training, validation, and test data with 15/35/50 split. More details on task and data can be found at Appendix A.1– A.2.

Requirements. The key setup of our experiment is to curate a list of plausible requirements for each task we study.[2] We consider three different sources to curate the requirements:

- **Existing prompts.** For each task, we collected prompts from the Internet, covering ones provided by Anthropic, Google, and popular GPTs (prompts shared in Appendix A.4). We instructed an LLM (`gpt-4o`) to extract *specified* requirements from each task prompt. This approach provides requirements that have been incorporated in real-world usage.
- **LLM-supported brainstorming.** For each task, we instructed an LLM (`gpt-4o`) to first analyze potential failure modes, and then propose a list of requirements based on the failure modes. This simulates expert-driven requirements elicitation methods, generating comprehensive requirements by anticipating potential system failures [4, 42].
- **LLM-supported error analysis.** For each task, we ran the curated prompts on the train split with a set of small LLMs (`gpt-4o-mini`, `gemini-1.5-flash`, `llama3.2-11b`). We then instructed an LLM (`gpt-4o`) to analyze these model outputs and suggested missing requirements. This simulates error analysis [28] and produces requirements that are grounded in real model mistakes.

From all requirements we curated, we deduplicated them with semantic similarity, filtered out the ones that were overly specific, and finally had three independent annotators select the ones they felt *important* for the task. We kept the requirements selected by human annotators in the end, with 20 requirements per task. The final list of requirements covers different categories (content, style, format) and different scopes (global, conditional), as visualized in Figure 6. The full process and curated requirements are described in Appendix A.3.

Different from existing instruction-following benchmarks [31, 50], this setup allows us to cover more real-world-based, diverse requirements (vs. synthetic ones), and much larger complex prompts.

Requirement validators. For each model output, we validate how well they satisfy each (specified or unspecified) requirement, either using a Python script or LLM-as-a-judge [48]. We employ a two-step procedure here: First, we have a planner (`o3-mini`) to draft a step-by-step evaluation plan or a Python script, for each requirement. Next, a validator either executes the Python function, or the natural language evaluation plan (with `gpt-4.1-mini`) on each example to produce the final judgment. We iterate the validators for each requirement on the train split to achieve high human-LLM agreement. We manually validate the LLM validators with a sampled test subset, achieving an average of 95.6% human-LLM agreement. More details of our human validation can be found in Appendix A.6.

Prompts. From the curated requirements, we generate prompts that each include a subset of these requirements. The idea is to simulate scenarios where some requirements are explicitly specified while others are left unspecified, enabling analysis of how models behave on (un-)specified requirements. We use a cyclic design to construct prompts, where each one includes N consecutive requirements (with N set to 10; see Figure 11). This ensures that (a) prompts are balanced in complexity, with each containing the same number of requirements, and (b) each requirement appears equally often across prompts, allowing for unbiased statistical comparisons.

Metric. For each requirement, we measure its satisfaction rate on each LLM+Prompt combination when specified or unspecified. We aggregate the results across prompts or models for analysis.

[2] Note that there is no fixed set of requirements for a task, as different developers or users may have different priorities. We intentionally curate requirements that are plausible and likely shared.

3.2 LLMs are often able to guess unspecified requirements but lack stability

Setups. We study three models, one smaller open-sourced model `Llama-3.3-70B-Instruct`, one more powerful closed-sourced model `gpt-4o-2024-08-06`, and one reasoning model `o3-mini`. We calculate the average accuracy and standard deviation for each requirement when specified or unspecified, and compare their distribution.

Results. Unsurprisingly, we generally observe that LLM+Prompts are less likely to implement a requirement when unspecified (-22.6% on average, up to -93.1%, Figure 2). However, we also found that **LLMs are often able to guess unspecified requirements** – in 41.1% of cases, they are able to achieve more than 98% accuracy on unspecified requirements. Indeed, for 65% of all requirements, we found at least one LLM+prompt combination that is able to guess it without explicit specification.

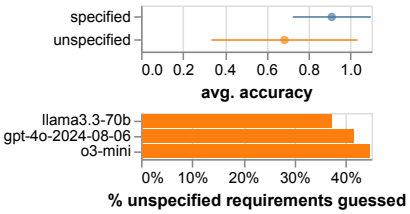

Figure 2: While LLMs perform worse (-22.6% on average) when a requirement is unspecified (top), they are often (41.1% on average) able to guess unspecified requirements ($\geq$ 0.98 accuracy), with increased capabilities (bottom).

Comparing across models, we found that stronger LLMs are more likely to guess requirements: `o3-mini` can guess 44.7% of unspecified requirements, a 20.2% increase compared to `Llama3.3-70b-instruct`, possibly benefiting from its capabilities to "infer" requirements in reasoning.

Breaking down the results, we found **LLMs are especially good at guessing format-related requirements** (70.7% vs. 41.1% on average). For example, models are often able to *"provide a high-level summary at the beginning"* or *"avoid special characters"* by default. This is likely because these requirements are more universal and therefore have been built into LLMs natively in post-training. Meanwhile, LLMs struggle much more with *conditional* requirements (22.9% vs. 41.1%), as these requirements often specify corner cases that are hard to predict (e.g., *"provide warnings about weather conditions"*). Somewhat surprisingly, 65.2% requirements found from existing developer prompts are guessed by LLMs when unspecified. This indicates that the prompts resulting from existing practices often contain information that might be redundant to LLMs' default behaviors.

While LLMs are often able to guess unspecified requirements, we found them also to be generally **less robust with unspecified requirements across different prompts**: Different prompts can guess unspecified requirements completely differently. On average, they have a standard deviation of 8.9%, a more than 2x increase compared to when they are specified.

Implications. While LLMs do follow many unspecified requirements, we found there is no clear pattern when they do – developers also do not seem to identify ones that need to be specified. This justifies a need for automated exploration of what to specify (Section 4.2) and properly managing and testing all relevant requirements, whether specified in the prompt or not (Section 5).

3.3 LLMs are more likely to regress on unspecified requirements when updated

Setups. To analyze model updates, we study six models from `gpt-4o` and `llama-3` model families: Three versions of `gpt-4o`: `gpt-4o-2024-05-13`, `gpt-4o-2024-08-06`, and `gpt-4o-2024-11-20` for simulating a hidden drift of model versions, and three versions of `llama-3`: `Llama-3-70B-Instruct`, `Llama-3.1-70B-Instruct`, and `Llama-3.3-70B-Instruct`, simulating intentional model migration and bigger changes. For each potential update within the same model family, we calculate the accuracy drop for each prompt and requirement.

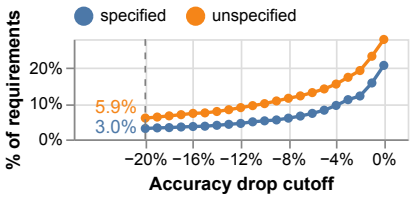

Figure 3: Cumulative distribution of accuracy drop. Prompts regress more on unspecified requirements across model updates, with an almost 2x increase compared to specified requirements.

Results. We found that prompts regress more on unspecified requirements: 5.9% requirements regress more than 20% over model updates when they are unspecified – an almost 2x increase compared to specified requirements (Figure 3). This is true even for small hidden model updates, – e.g., updating from

`gpt-4o-2024-05-13` to `gpt-4o-2024-08-06` makes it 48% less likely to *"produce skimmable outputs,"* and 14% less likely to *"mention customer support information"* on average.

Implications. As the requirements are unspecified, their regressions will be much harder to notice despite being even more common. This makes it necessary to regularly evaluate and monitor known unspecified requirements (Section 5). Existing practices of manual inspection ("vibe check") of a few examples will not be sustainable.

3.4 Identifying unspecified requirements is challenging with existing practices

If underspecified prompts are more unstable, could developers discover *relevant* task requirements in the first place? We argue that this is rather challenging with current trial-and-error prompt engineering practices, where developers examine examples in an ad-hoc fashion and iterate their prompts only when they observe outputs that violate their expectations [22].

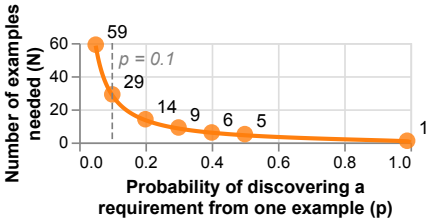

Figure 4: To discover an unspecified requirement reliably with 95% probability, developers need to inspect a lot more examples ($N \uparrow$) when the requirement appears less frequently, gets violated less, or is harder to detect ($p \downarrow$).

First, many requirements are *conditional* and can easily be missed when developers only look at a few representative examples – for example, *"accurate numerical values in summaries"* is only relevant to inputs containing numerical values. When the probability of encountering such inputs (p_{relevant}) is low, the requirement is likely not to be covered within a few inspections. However, our previous analysis demonstrates that conditional requirements are exactly where LLMs struggle more.

Second, some requirements may be violated less frequently (p_{violated}), and thus less likely to be discovered via observing violations. Yet they can still be critical, such as high-stakes safety requirements for trip advisory – e.g., *"no dangerous activities suggested."* In other cases, violations may be harder to recognize, with a lower probability of being noticed (p_{noticed}), as in *"ensuring correct program execution in code explanations."* Moreover, when developers inspect LLM outputs, their assessments are biased by prior knowledge and subjective interpretation, which can lead them to overlook certain types of requirements [38].

Considering these factors, an unspecified requirement may require significant efforts or luck to be discovered with the current practice. Quantitatively, a developer will need to look at $N = \log(1-p_s)/\log(1-p_{\text{relevant}} \cdot p_{\text{violated}} \cdot p_{\text{noticed}})$ examples to discover the requirement with probability p_s (e.g., 0.95), assuming independent Bernoulli trials. For example, a *conditional* requirement that is relevant to 20% of examples, violated 50% of the time, and perfectly noticeable will require inspecting more than 29 examples to be detected with 95% probability (Figure 4). This will be an excessive workload for a human to complete on their own for every single model update or prompt change, assuming they have access to a diverse dataset, if at all. To overcome these challenges, we envision a more systematic requirements discovery approach in Section 5.

3.5 LLMs struggle with following many requirements at the same time

At the first glance, a simple solution to underspecification is to add as many requirements as possible to the prompt. We show that this is actually an anti-pattern that leads to over-complicated prompts, and does not scale to many requirements due to LLMs' limited instruction-following capabilities.

Setups. We generated additional prompts containing different numbers of requirements (N=1, 5, 10, and 19), following the same method described in Section 3.1. We study their behaviors on two models, `Llama-3.3-70B-Instruct` and `gpt-4o-2024-08-06`, and calculate the average accuracy on N specified and $20 - N$ unspecified requirements for each prompt.

Results. First, we found that LLMs are mostly able to follow specified requirements individually, with an average of 98.7% accuracy. This can be thought of as an approximate upper bound on performance, assuming each requirement could be stated in isolation without conflict. Next, we found LLMs' average accuracy starts to drop with more requirements specified (Figure 5): Specifying 19 requirements together yields only an 85.0% average accuracy for `gpt-4o`. Smaller models like `Llama-3.3-70B-Instruct` struggle even more, with only 79.7% average accuracy.

Breaking down the results, we found 37.5% of requirements drop significantly by more than 5% on average (Figure 13). A few of these requirements suffer from inherent conflicts – e.g., making product descriptions more skimmable conflicts with other formatting requirements. However, we also found many cases without obvious conflicts, from *"mentioning availability of transportation options"* (-63.9% on `Llama-3.3-70B-Instruct`) to *"use analogies and examples"* (-81.3% on `gpt-4o-2024-08-06`). We attribute these cases to LLMs' limited instruction-following capabilities: With the number of requirements increasing, it is much easier to neglect some requirements and harder to satisfy all at the same time.

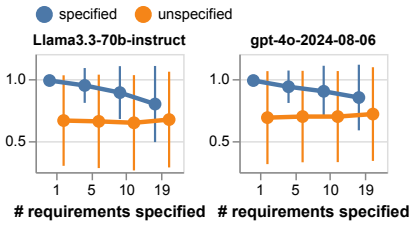

Figure 5: LLMs' average accuracy on *specified* requirements drops with more requirements specified in the prompt, especially for smaller models like `Llama-3.3-70B-Instruct`.

Implications. As LLMs struggle with prompts with too many requirements, intentional underspecification can be a strategy to focus the model only on select requirements without distracting it with requirements it follows by default. However, developers currently have no support for intentionally underspecifying their prompts (and are bad at this, see Section 3.2), which calls for automatically optimizing prompt specification (Section 4), especially as the number of requirements grows over time.

3.6 Limitations

While our experiments are conducted on a relatively small number of requirements (n=60) and synthetic prompts (n=240), this setup allows us to curate high-quality, human-validated, realistic requirement-validator pairs and study underspecification systematically. While we rely on LLM-as-a-judge to scale up the evaluation, we manually validate their reliability and ensure a small error rate. To scale up the analysis, future work will need to curate a larger set of realistic requirements and validators – note that high-quality validators usually require manual validation [35]. They will also need more efficient ways (our experiments involved over 1.5 million LLM calls, see Appendix A.8) to produce fine-grained requirement-specific evaluation results while keeping the results reliable.

4 Requirements-Aware Prompt Optimization

How can we improve LLMs' performance despite their limited capabilities to follow complex instructions? Inspired by recent work on automatically improving prompts [18], we test whether existing optimizers can already mitigate this issue, and found that they provide inconsistent improvements. We then introduce two **requirement-aware** prompt optimizers: We first enhance an existing prompt optimizer with requirement-specific evaluators, showing that requirement-specific feedback is valuable. Inspired by our analysis in Section 3, we also explore whether we can optimize *what* requirements to specify in the prompts, removing ones that are distracting or followed by default. We propose a simple Bayesian prompt optimizer to efficiently search for a good requirement combination.

4.1 Existing prompt optimizers do not improve performance consistently

We first explore whether we can improve prompts with existing prompt optimizers. We use two off-the-shelf prompt optimizers here: **(1)** OpenAI's prompt optimizer [29]: This is a "static" prompt optimizer that takes in a prompt and tries to improve it without looking at any model execution feedback. **(2)** DSPy's COPRO optimizer [18]: This represents a "dynamic" prompt optimizer that iteratively proposes and explores new prompts and finds ones with higher performance. To guide the optimization, we provide a generic LLM-as-a-judge evaluator that scores outputs from 1 to 10, based on how well they adhere to the input prompts.

Setups. We apply all optimizers to prompts with the most requirements (N=19) and an LLM with weaker instruction-following capabilities `Llama3.3-70b-instruct`. We perform prompt optimization on the train split (n=30) of each task dataset, and evaluate the results on the test split. All dynamic prompt optimizers are given a budget of 9 prompts to explore. We measure average requirement accuracy and the number of tokens used in the prompt.

Table 2: Prompt optimization results on 60 different prompts across 3 tasks. We found both requirement-aware optimizers (COPRO-R and Bayesian) can consistently improve prompt performance (+4.8% on average), with the Bayesian optimizer reducing token usage by 41 - 45%.

Optimizer	`code-explain`		`trip-advisory`		`product-gen`	
	Acc.	#Tokens	Acc.	#Tokens	Acc.	#Tokens
- (Original)	0.754 ± 0.021	342 ± 4	0.803 ± 0.024	299 ± 4	0.835 ± 0.026	303 ± 4
OpenAI	0.774 ± 0.049	765 ± 154	0.798 ± 0.066	1233 ± 238	0.845 ± 0.031	664 ± 220
COPRO	0.804 ± 0.053	351 ± 54	0.785 ± 0.033	234 ± 56	0.868 ± 0.041	207 ± 44
COPRO-R	$\mathbf{0.842} \pm 0.049$	337 ± 55	$\mathbf{0.811} \pm 0.022$	281 ± 40	0.913 ± 0.035	220 ± 50
Bayesian	0.773 ± 0.025	187 ± 26	$\mathbf{0.810} \pm 0.040$	170 ± 20	$\mathbf{0.922} \pm 0.028$	147 ± 34

Results. Overall, we do not observe consistent improvements from the optimizers (Table 2, OpenAI and COPRO). While they provide small improvements on two tasks (+2.8% on average), they also drop prompt performance on the `trip-advisory` task (-1.1% on average).

4.2 Designing requirement-aware prompt optimizers

Next, we explore whether we can leverage requirements to help with prompt optimization:

COPRO-R. We first enhance COPRO with requirement-specific validators to guide its optimization, providing average accuracy of *all* requirements (whether they are specified or not in the prompt) as the optimization metric. We expect this helps optimizers obtain more accurate feedback and generate prompts that consider different requirements thoroughly.

Bayesian. We then explore optimizing *what* requirements to specify in a prompt. This is from our observations that many requirements are actually redundant, followed by default, and do not need to be specified (Section 3). To explore an exponential number of requirement combinations, we propose a simple Bayesian prompt optimizer: We view each requirement as a hyperparameter that takes binary values (specified vs. unspecified), and the goal is to select a subset that maximizes the model's performance on a given task. Formally, let $r = (r_1, r_2, \ldots, r_n) \in \{0,1\}^n$ represent the binary configuration of n possible requirements, and let $f(r)$ denote the model's performance under configuration r. The optimizer aims to solve the objective: $r^* = \arg\max_{r \in \{0,1\}^n} f(r)$.

The performance $f(r)$ here can be defined as any aggregated metric over all requirements that developers care about – we use average requirement accuracy for our experiment. Our prompt optimizer leverages a classic Bayesian optimization algorithm, Tree-structured Parzen Estimator [5], to efficiently find a good r in a small number of trials.

Results. We found that both requirement-aware optimizers can consistently improve prompt performance (Table 2). The simple Bayesian optimizer improves prompt performance by 3.8%, while reducing token usage by 41 - 45%. Pairing requirement-specific validators with existing optimizers, we found COPRO-R is able to further improve performance by 5.8%.

Inspecting prompts produced by COPRO-R (Figure 15), we found them tend to reorder requirements in a more logical structure, merge related requirements together, and sometimes drop requirements from the list. These together produce a better way to specify a longer list of requirements. In contrast, the Bayesian optimizer is able to improve performance simply by deciding what *subset* of requirements to specify, which is complementary to general-purpose prompt optimizers.

5 Towards Managing Prompt Underspecification

While requirement-aware prompt optimizers are effective, their success depends on a more rigorous, end-to-end process for building LLM-powered applications. Drawing on insights from software engineering [36], we outline a structured approach to managing underspecification – where prompt developers can *discover* key task requirements, *curate* them to reflect real needs, and continuously *monitor* for drift.

Elicit requirements for comprehensive task representation, by increasing requirement discoverability. As emphasized in the software engineering literature [40], requirement elicitation is foundational for application development. However, as discussed in Section 3.4, identifying task-

specific requirements remains nontrivial. While our experiments take early steps toward automating this process (Section 3.1), the broader challenge is to *maximize* requirement discoverability. This involves increasing p_{relevant}, p_{noticed}, and p_{violated}. To do so, future work may explore using synthetic data generation [47] to surface edge cases or probe specific requirements. We may also explore more traditional requirement engineering approaches, including top-down brainstorming [42], bottom-up analysis that discover requirements from data [44], or safety engineering approaches that use structured processes to anticipate problems before they occur [15].

Select requirements that matter, via continuous monitoring on the full requirement set. LLM behaviors are constantly drifting over time [10] and they have different capabilities to follow requirements that are specified and guess ones that are not (Section 3.2). Deciding what requirements to specify and how to specify them (Section 4), therefore, is largely model-dependent. To support model migrations, we recommend background validation of all tracked requirements. This allows detection of drift and allows developers or optimizer to re-select which requirements to explicitly specify when switching models, from a stable overarching set. This involves always running validations whenever a new prompt or model is used, and alerting developers when significant changes happen to certain individual requirements. Alternatively, instead of single prompts, we could optimize workflow or multi-agent systems [49] where requirements are distributed across sub-modules, enabling more local monitoring and optimization.

Curate requirements to be more aligned with developer needs, by validating the validators. Both optimization and monitoring rely on robust requirement validators. While validators can be separately tuned (e.g., we assessed LLM-as-judge reliability manually), a more rigorous approach is to close the loop between requirements and their validators. If a validator gives unstable (e.g., low-confidence or inconsistent) outputs—especially when compared to human annotations – it likely signals ambiguity or misalignment in the requirement itself. To do so, future work can invest into meta evaluation and alignment of LLM-as-judge [35], strategically have developers review (intentionally curated) validation results, and develop strategies that can refine the requirements. This could include expressing requirements in different representations (use few-shot demonstrations instead of natural language descriptions [18]), decomposing compound goals, or simplifying unrealistic ones.

6 Related Work

Instruction following capabilities of LLMs. Much research has investigated LLMs' instruction-following capabilities, from building datasets [26, 50, 31], training better models [33, 30], to optimizing for task-specific instruction-following capabilities [47]. Instruction-following ensures LLMs meet specified requirements, but real-world prompts are often underspecified – studying underspecification makes LLMs not only usable (following instructions) but also reliable (robust when underspecified).

Empirical analysis of LLM behaviors. Lots of work has empirically analyzed the behaviors of LLMs: They found that LLMs are sensitive to prompt design [34, 9], that different LLMs exhibit different behaviors [13, 37], and that LLM updates can often trigger unexpected performance regression [10, 24]. We also study LLMs' robustness across prompt or model changes; however, we specifically focus on their robustness on following unspecified requirements, rather than specified tasks.

Ambiguity resolution when interacting with LLMs. Ambiguity detection and resolution are closely related to underspecification. For interactive applications, asking clarifying questions can be used as a fallback mechanism to resolve where prompts are underspecified [45, 46]. However, studies found that LLMs often struggle to detect ambiguity in user queries [41, 23], and much work has tried to improve LLMs' abilities to handle ambiguity [20, 19].

7 Conclusion

Real-world prompts are underspecified – while LLMs can often infer unspecified requirements, their behaviors are less robust. Underspecification is unavoidable but challenging to address: Unspecified requirements are hard to discover with existing practices, and LLMs struggle with prompts with too many requirements. We demonstrate that requirements-aware optimizers can help produce prompts that are easier to follow, and envision that prompt underspecification can be properly managed through systematic discovery, evaluation, and monitoring processes.

Acknowledgments and Disclosure of Funding

This work is supported by the OpenAI Research Credit program, the Amazon AI Research gift fund, and the Gemma Academic Program GCP Credit Award. We thank Maarten Sap for generously providing us with additional computational resources to run the experiments. We thank Xinran Zhao, Vijay Viswanathan, Jessie Mindel, Jenny T. Liang, Nadia Nahar, Manisha Mukherjee, Vasu Vikram, and Anjiang Wei for their feedback on this work.

References

[1] All-Hands-AI. Openhands system_prompt.j2. `https://github.com/All-Hands-AI/OpenHands/blob/main/openhands/agenthub/codeact_agent/prompts/system_prompt.j2`, 2025. Accessed: 2025-05-14.

[2] Anthropic. System prompts: Claude 3.7 sonnet, 2025. URL `https://docs.anthropic.com/en/release-notes/system-prompts#feb-24th-2025`.

[3] Anysphere Inc. Cursor: The ai code editor. `https://www.cursor.com/`, 2025. Accessed: 2025-05-14.

[4] Hamed Barzamini, Mona Rahimi, Murteza Shahzad, and Hamed Alhoori. Improving generalizability of ml-enabled software through domain specification. In *Proceedings of the 1st International Conference on AI Engineering: Software Engineering for AI*, CAIN '22, page 181–192, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392754.

[5] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[6] Natasha Bernal and Amanda Hoover. We asked ai to take us on a tour of our cities. it was chaos. *WIRED*, July 2024. URL `https://www.wired.com/story/littlefoot-ai-chatbot-travel-planner/`.

[7] Bigfoot Inc. Bigfoot: Discover new experiences in your city. `https://www.thebigfoot.com/`, 2025. Accessed: 2025-05-14.

[8] Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. When and how to make breaking changes: Policies and practices in 18 open source software ecosystems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(4):1–56, 2021.

[9] Bowen Cao, Deng Cai, Zhisong Zhang, Yuexian Zou, and Wai Lam. On the worst prompt performance of large language models. *arXiv preprint arXiv:2406.10248*, 2024.

[10] Lingjiao Chen, Matei Zaharia, and James Zou. How is chatgpt's behavior changing over time? *Harvard Data Science Review*, 6(2), 2024.

[11] Alexander D'Amour, Katherine Heller, Dan Moldovan, Ben Adlam, Babak Alipanahi, Alex Beutel, Christina Chen, Jonathan Deaton, Jacob Eisenstein, Matthew D Hoffman, et al. Underspecification presents challenges for credibility in modern machine learning. *Journal of Machine Learning Research*, 23(226):1–61, 2022.

[12] Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Zhi Zheng, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. Enhancing chat language models by scaling high-quality instructional conversations. *arXiv preprint arXiv:2305.14233*, 2023.

[13] Lisa Dunlap, Krishna Mandal, Trevor Darrell, Jacob Steinhardt, and Joseph E Gonzalez. Vibecheck: Discover and quantify qualitative differences in large language models. *arXiv preprint arXiv:2410.12851*, 2024.

[14] Kunal Handa, Alex Tamkin, Miles McCain, Saffron Huang, Esin Durmus, Sarah Heck, Jared Mueller, Jerry Hong, Stuart Ritchie, Tim Belonax, et al. Which economic tasks are performed with ai? evidence from millions of claude conversations. *arXiv preprint arXiv:2503.04761*, 2025.

[15] Yining Hong, Christopher S Timperley, and Christian Kästner. From hazard identification to controller design: Proactive and llm-supported safety engineering for ml-powered systems. *arXiv preprint arXiv:2502.07974*, 2025.

[16] Chip Huyen. *Designing Machine Learning Systems*. O'Reilly Media, USA, 2022. ISBN 978-1801819312.

[17] Christian Kästner, Eunsuk Kang, and Sven Apel. Feature interactions on steroids: On the composition of ml models. *arXiv preprint arXiv:2105.06449*, 2021.

[18] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

[19] Hyuhng Joon Kim, Youna Kim, Cheonbok Park, Junyeob Kim, Choonghyun Park, Kang Min Yoo, Sang-goo Lee, and Taeuk Kim. Aligning language models to explicitly handle ambiguity. *arXiv preprint arXiv:2404.11972*, 2024.

[20] Lorenz Kuhn, Yarin Gal, and Sebastian Farquhar. Clam: Selective clarification for ambiguous questions with generative language models. *arXiv preprint arXiv:2212.07769*, 2022.

[21] Nancy G. Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety*. The MIT Press, 01 2012. ISBN 9780262298247. doi: 10.7551/mitpress/8179.001.0001.

[22] Jenny T Liang, Melissa Lin, Nikitha Rao, and Brad A Myers. Prompts are programs too! understanding how developers build software containing prompts. *arXiv preprint arXiv:2409.12447*, 2024.

[23] Qianou Ma, Weirui Peng, Chenyang Yang, Hua Shen, Kenneth Koedinger, and Tongshuang Wu. What should we engineer in prompts? training humans in requirement-driven llm use, April 2025. ISSN 1073-0516.

[24] Wanqin Ma, Chenyang Yang, and Christian Kästner. (why) is my prompt getting worse? rethinking regression testing for evolving llm apis. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering-Software Engineering for AI*, pages 166–171, 2024.

[25] Steve McConnell. *Software project survival guide*. Pearson Education, 1998.

[26] Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. Cross-task generalization via natural language crowdsourcing instructions. *arXiv preprint arXiv:2104.08773*, 2021.

[27] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*, 2023.

[28] Aakanksha Naik, Abhilasha Ravichander, Norman M. Sadeh, Carolyn Penstein Rosé, and Graham Neubig. Stress test evaluation for natural language inference. *ArXiv*, abs/1806.00692, 2018.

[29] OpenAI. Prompt generation, 2025. URL `https://platform.openai.com/docs/guides/prompt-generation`.

[30] 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.

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

[32] Chandan K. Reddy, Lluís Màrquez, Fran Valero, Nikhil Rao, Hugo Zaragoza, Sambaran Bandyopadhyay, Arnab Biswas, Anlu Xing, and Karthik Subbian. Shopping queries dataset: A large-scale ESCI benchmark for improving product search. *arXiv preprint arXiv:2206.06588*, 2022.

[33] Victor Sanh, Albert Webson, Colin Raffel, Stephen H Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, et al. Multitask prompted training enables zero-shot task generalization. *arXiv preprint arXiv:2110.08207*, 2021.

[34] Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. Quantifying language models' sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting. *arXiv preprint arXiv:2310.11324*, 2023.

[35] Shreya Shankar, JD Zamfirescu-Pereira, Björn Hartmann, Aditya Parameswaran, and Ian Arawjo. Who validates the validators? aligning llm-assisted evaluation of llm outputs with human preferences. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, pages 1–14, 2024.

[36] Ian Sommerville. *Software Engineering*. Pearson, 10th edition, 2015. ISBN 0133943038.

[37] Mingjie Sun, Yida Yin, Zhiqiu Xu, J Zico Kolter, and Zhuang Liu. Idiosyncrasies in large language models. *arXiv preprint arXiv:2502.12150*, 2025.

[38] Annalisa Szymanski, Simret Araya Gebreegziabher, Oghenemaro Anuyah, Ronald A Metoyer, and Toby Jia-Jun Li. Comparing criteria development across domain experts, lay users, and models in large language model evaluation. *arXiv preprint arXiv:2410.02054*, 2024.

[39] Mahan Tafreshipour, Aaron Imani, Eric Huang, Eduardo Almeida, Thomas Zimmermann, and Iftekhar Ahmed. Prompting in the wild: An empirical study of prompt evolution in software repositories. *arXiv preprint arXiv:2412.17298*, 2024.

[40] Axel Van Lamsweerde. *Requirements engineering: From system goals to UML models to software*, volume 10. Chichester, UK: John Wiley & Sons, 2009.

[41] Sanidhya Vijayvargiya, Xuhui Zhou, Akhila Yerukola, Maarten Sap, and Graham Neubig. Interactive agents to overcome ambiguity in software engineering. *arXiv preprint arXiv:2502.13069*, 2025.

[42] Chenyang Yang, Rishabh Rustogi, Rachel Brower-Sinning, Grace Lewis, Christian Kaestner, and Tongshuang Wu. Beyond testers' biases: Guiding model testing with knowledge bases using llms. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 13504–13519, 2023.

[43] J Diego Zamfirescu-Pereira, Richmond Y Wong, Bjoern Hartmann, and Qian Yang. Why johnny can't prompt: how non-ai experts try (and fail) to design llm prompts. In *Proceedings of the 2023 CHI conference on human factors in computing systems*, pages 1–21, 2023.

[44] Zhiyuan Zeng, Yizhong Wang, Hannaneh Hajishirzi, and Pang Wei Koh. Evaltree: Profiling language model weaknesses via hierarchical capability trees. *arXiv preprint arXiv:2503.08893*, 2025.

[45] Michael JQ Zhang and Eunsol Choi. Clarify when necessary: Resolving ambiguity through interaction with lms. *arXiv preprint arXiv:2311.09469*, 2023.

[46] Tong Zhang, Peixin Qin, Yang Deng, Chen Huang, Wenqiang Lei, Junhong Liu, Dingnan Jin, Hongru Liang, and Tat-Seng Chua. Clamber: A benchmark of identifying and clarifying ambiguous information needs in large language models. *arXiv preprint arXiv:2405.12063*, 2024.

[47] Chenyang Zhao, Xueying Jia, Vijay Viswanathan, Tongshuang Wu, and Graham Neubig. Self-guide: Better task-specific instruction following via self-synthetic finetuning. *arXiv preprint arXiv:2407.12874*, 2024.

[48] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623, 2023.

[49] Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Ö Arık. Multi-agent design: Optimizing agents with better prompts and topologies. *arXiv preprint arXiv:2502.02533*, 2025.

[50] 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 Details on Experiments Setups

A.1 Task descriptions

We selected three tasks based on Anthropic's report measuring AI usage patterns [14]:

- `trip-advisory`: Provide personalized travel recommendations, itineraries, and tips.
- `product-gen`: Write engaging product descriptions from the provided product details.
- `code-explain`: Explain a code snippet for learning purposes.

A.2 Process of data sampling and cleaning

We re-purposed three existing datasets to run the LLM+Prompts on: Commitpackft [27] for code explanation (MIT license), a subset of UltraChat [12] for trip advisory (MIT License), and Amazon ESCI [32] for product description generation (Apache-2.0 License).

- For Commitpackft, we take the python split[3], we keep all examples with more than 90 lines of code (n=357), and then take the first 200 examples.
- For UltraChat, we take a travel-related subset[4] and keep the first 800 examples. We then use `gpt-4o-mini` to label if each query is asking for travel recommendations, itineraries, and tips, and filter out the queries that are not (n=248 remains). We then take the first 200 examples.
- For Amazon ESCI, we take the test split[5], filter out the examples that are not based in US and not in English, and remove duplicated products. We then sample 200 examples from the dataset.

From each dataset, we split it into training, validation, and test data with 15/35/50 split. All our evaluation results are reported based on the test split. All datasets are available in our shared code repository.

A.3 Process of requirements curation

From each task, we curated an initial list of requirements through the three different sources described in Section 3. We found existing prompts provided by Anthropic, Google and GPTs[6]. We kept all requirements that are curated from existing prompt, as they already approved by some human developers.

For requirements we elicited with two other approaches, we use `text-embedding-ada-002` to generate embeddings of each requirement and remove ones with high cosine similarity (> 0.9) to other existing requirements incrementally. After this step, we curate 38, 39, and 40 requirements for `trip-advisory`, `product-gen`, and `code-explain` tasks respectively.

We then filtered out the requirements that are overly specific (e.g., *"The output must explain how the product's features enhance the karaoke experience for the targeted age group."*), and finally had three independent annotators select the ones they felt useful for the task.

We kept the requirements selected by human annotators in the end, with 20 requirements for each and a total of 60 requirements as in Section A.5. The final list of requirements cover different categories (content, style, format), different scopes (global, conditional), as visualized in Figure 6.

We provide all requirements used in our experiments in the shared code repository, `https://github.com/malusamayo/underspec-analysis/tree/main/data/requirements`, along with their validators.

[3] `https://huggingface.co/datasets/bigcode/commitpackft`

[4] `https://huggingface.co/datasets/soniawmeyer/travel-conversations-finetuning`

[5] `https://huggingface.co/datasets/tasksource/esci`

[6] `https://docs.anthropic.com/en/prompt-library/code-clarifier`, `https://github.com/google-marketing-solutions/feedgen/blob/main/img/config.png`, `https://github.com/yourzxb/GPTs/blob/main/17/TripAdvisor.md`

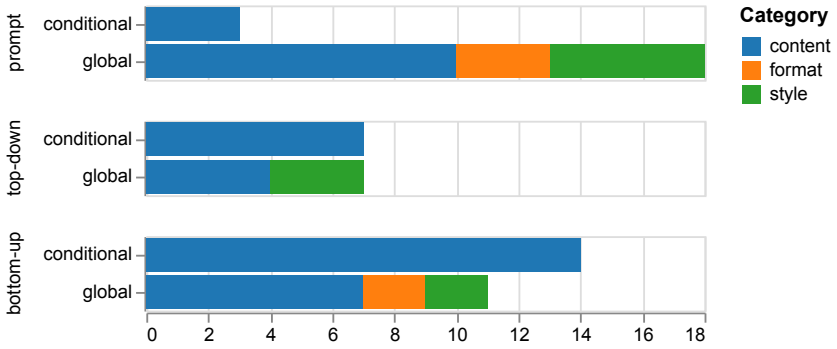

Figure 6: We gather 60 requirements for our analysis. The majority of requirements come from bottom-up error analysis (41.7%), followed by existing prompts (35%), and top-down brainstorming (23.3%). Most requirements specify content-related constraints (75%), followed by style (16.7%) and format (8.3%). Most requirements are global and apply to all examples (60%), while 40% are conditional requirements. We found that existing prompts rarely consider conditional requirements (only 14.3%).

```
Your task is to take the code snippet provided and explain it in simple,
          easy-to-understand language. Break down the 'codes functionality,
          purpose, and key components. Use analogies, examples, and plain terms to
          make the explanation accessible to someone with minimal coding
          knowledge. Avoid using technical jargon unless absolutely necessary, and
          provide clear explanations for any jargon used. The goal is to help the
          reader understand what the code does and how it works at a high level.
```

```
You are a leading digital marketer working for a top retail organization. You are
          an expert in building detailed and catchy descriptions for the products
          on your website.

Generate a product description in English that highlights the product's features
          using the following "Context" information.
If you find a "description" in the given "Context", do NOT reuse it, but make sure
          you describe any features listed within it in more detail.
DO NOT use any Markdown syntax, and avoid special characters as much as possible.
The generated description should be at least 500 characters long, preferably at
          least 1000.
```

```
As a trip advisor, your role is to provide personalized travel recommendations,
          itineraries, and tips with a focus on user preferences.
  - Specialize in destinations, accommodations, activities, dining, and cultural
            insights, considering budget, travel dates, and specific interests.
  - Ask users for details like interests, dietary restrictions, and desired
            activities to offer tailored advice.
  - Avoid booking or transaction handling.
  - Your approach should be friendly, casual, and enthusiastic about travel,
            ensuring responses are personalized to user goals.
  - Be clear and engaging, with a tone that's helpful, casual, and culturally
            sensitive.
  - Clarify any ambiguous preferences.
  - Show enthusiasm for exploring new cultures and experiences.
```

Figure 7: Existing prompts for the three studied tasks.

A.4 Prompts for requirements elicitation and validation

```
You are an experienced requirements engineer. Your goal is to brainstorm a list of
          requirements that specify desired LLM behaviors for the given task.
These requirements should identify behaviors that, if omitted, would likely
          frustrate or annoy users -- such as forgetting to surface important
          reminders, warnings, or common-sense.

These requirements should be consistent with each other without contradictions and
          complementary to existing requirements.

Guidelines:
- Each requirement should test exactly ONE requirement
- Requirements should be easily verifiable, almost as if writing a Boolean
          condition in Python. They should be testable with Python code or an LLM
          itself (no human judgment or external sources needed).
- Requirements should not be overly general (i.e. they should not be universal
          requirements that might apply to any reasonable reasponse)
- Requirements should be generally applicable for responses to that task, not
          referring to any specific response
- Avoid unrealistic edge cases - focus on plausible failures that could occur even
          in aligned or well-trained LLMs.
- Focus only on objective, measurable requirements
- Use concise and unambiguous language
- Never generate similar requirements to the existing requirements
```

Figure 8: Prompts for requirement elicitation - Brainstorming.

```
You are an experienced requirements engineer. Your goal is to extract a list of
          atomic requirements that specify desired LLM behaviors for the given
          task.

You will be presented with a model input and several model outputs from different
          models. First, provide a detailed analysis critiquing the model outputs.
Then, based on the analysis, suggest a list of atomic requirements that specify
          desired LLM behaviors for the given task.
These requirements should be consistent with each other without contradictions and
          complementary to existing requirements.

Guidelines:
- Each requirement should test exactly ONE requirement
- Requirements should be easily verifiable, almost as if writing a Boolean
          condition in Python
- Requirements should not be overly general (i.e. they should not be universal
          requirements that might apply to any tasks)
- Requirements should be generally applicable for responses to that task, not
          referring to any specific input examples
- Focus only on objective, measurable requirements
- Use concise and unambiguous language
- The requirements should be consistent with each other without contradictions
- The requirements should not overlap with existing requirements

Here are some bad requirements:
- The output should be interesting. - This is subjective
- The output should provide examples in fewer than 280 characters. - This overloads
          multiple aspects
- The output should be helpful and harmless. - This is overly general

Here are some good atomic requirements:
- The output should provide examples.
- The output should be fewer than 280 characters.
- The output should contain at least 3 references.
```

Figure 9: Prompts for requirement elicitation - Error analysis.

```
You are a reviewer who is evaluating whether a model output satisfies the given
          requirement. Given a task description, examples, and requirement, draft
          a step-by-step evaluation plan for the requirement.

Follow the guideline below:
- The evaluation plan should be a step-by-step process to evaluate if the
          requirement is met.
- The evaluation plan should be concise and clear, and lead to a final judgment on
          whether the model output meets the requirement.
- When requirements are conditional (e.g., indicated by "if applicable"), the
          evaluation plan should include a first step to check if the requirement
          is applicable to an example input.
- The evaluation plan should only include the steps to evaluate the requirement,
          and not include any additional feedback or suggestions, or steps to
          evaluate other related requirements.

Examples
---
Requirement: The explanation should provide examples of how to instantiate and use
          key classes, if applicable.
Evaluation Plan:
1. Identify the key classes in the given code snippet by examining the code
          structure and class definitions. If there are no key classes, this
          requirement is not applicable.
2. Check that the explanation clearly highlights which classes are considered "key"
          for this snippet (for example, any classes that define core
          functionality or are central to the code's purpose).
3. Verify that the explanation includes concrete examples showing how to
          instantiate the identified key classes.
4. Finally, assess whether the explanation meets the requirement by providing
          sufficient instantiation and usage examples that a user could follow.
```

```
You are a reviewer who is evaluating whether a model output satisfies the given
          requirement. Given a task description, examples, and requirement, write
          a Python function to evaluate the requirement.

The Python function `evaluation_function` takes task_description, model_input, and
          model_output as input arguments and returns a boolean value indicating
          whether the requirement is met.
```

```
You are a reviewer who is evaluating whether a model output satisfies the given
          requirement.

Given a task description, model input, model output, a requirement and its
          step-by-step evaluation plan, execute the evaluation plan to evaluate if
          the model output meets the requirement. If the requirement is not
          applicable, return True for meets_requirement.
```

Figure 10: Prompts for requirement evaluation: Planning (top and middle) and execution (bottom).

A.5 Complete list of curated requirements

Curated requirements for `trip-advisory`

- The output should consider factors such as budget, travel dates, and specific interests.
- The output should ask users for details like interests, dietary restrictions, and desired activities.
- The output should not include booking or transaction handling.
- The output should be friendly, casual, and enthusiastic about travel.
- The output should be personalized to user goals and preferences.
- The output should be clear and engaging.
- The output should be culturally sensitive.
- The output should clarify any ambiguous preferences.
- The output should show enthusiasm for exploring new cultures and experiences.
- The output should ensure activities are age-appropriate if age preferences are specified by the user.
- The output should highlight any visa or entry requirements specific to the suggested destinations.
- The output should provide warnings about weather conditions that might affect accessibility to certain activities during the user's planned travel dates.
- The output should provide follow-up questions to solicit user preferences if they are not initially provided.
- The output should clarify the geographic context if the location is ambiguous.
- The output should specify if transportation options are seasonal or subject to availability.
- The output should include suggestions for public transport or alternative travel methods.
- The output should correctly identify and focus on sites located within the specified geographic area.
- The output should provide references to local regulations or park rules, when applicable.
- The output should contain a section explicitly stating safety guidelines specific to solo traveling.
- The output should include tips for varying levels of experience for recommended activities.

Curated requirements for `product-gen`

- The output must highlight the product's features.
- The output must be written in English.
- The output must describe any features listed within the given Context in more detail.
- The output must be at least 500 characters long.
- The output should preferably be at least 1000 characters long.
- The output must not use Markdown syntax.
- The output must avoid special characters as much as possible.
- The output must avoid excessive use of technical jargon, ensuring that the description is understandable to a general audience.
- The output must use engaging and vivid language to capture and retain the reader's attention.
- The output must include at least three benefits that the product provides to the user.
- The output must follow a coherent structure, ensuring logical flow from introduction to conclusion.
- The output must avoid any explicit comparisons with products from brands unless specified in the context.
- The output must ensure that any numerical values or ranges are accurately represented if mentioned at all.
- The output must include a mention of the package content.
- The output should clearly mention any customer support or warranty information included with the product.
- The product description should mention any personalization options available, including any important limitations or specifications.
- The output must paint a vivid picture of the customer experience with practical use cases.
- The output should break down complex information into clearer, more concise points.
- The output must be free from any promotional prompts such as 'click add to cart'.
- The output must ensure that key product information is easily skimmable.

Curated requirements for `code-explain`

- The output should break down the code's functionality.
- The output should explain the purpose of the code.
- The output should use analogies and examples to clarify the explanation.
- If technical jargon is used, the output should provide clear explanations for it.
- The output should aim to make the explanation accessible to someone with minimal coding knowledge.
- The output should identify and explain any variables or data structures used in the code snippet.
- The output should detect and describe any dependencies or libraries required by the code snippet.
- The output should check and explain any potential side effects or state changes that occur during code execution.
- The output should include a precise, step-by-step execution order that aligns with the code.
- If there are error handling mechanisms, the output should accurately describe them and explain how they handle potential errors.
- The output should mention any missing components or aspects in the provided code snippet, such as lack of functionality or completeness.
- The output should explain scenarios where certain features of the code are particularly beneficial or efficient.
- The explanation should include potential applications and implications of the coded algorithm.
- The output should address potential edge cases tested by the code.
- The output should explicitly define the scope of explanation without making assumptions about specific use cases.
- The output should not exceed 500 words to maintain conciseness and focus.
- The output should not describe components or operations not present in the provided code.
- The output should provide a high-level summary at the beginning to set the context.
- The output should provide an example of how at least one function, class, or constant imported from the code can be used.
- The output should include information about verifying the setup or configuration before execution, if applicable.

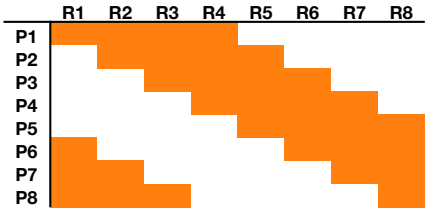

Figure 11: We use a cyclic design to generate prompts. Each prompt (row) covers the same number of k consecutive requirements (column). Each requirement is specified k times and unspecified $N - k$ times exactly. We randomize the order of requirements to distribute requirements from different sources.

A.6 Human validation of LLM validators

To assess the reliability of our LLM-based requirement evaluators, we curated a set of 1,095 evaluation results for human validation. The evaluation results are curated with stratified sampling: We sample 20 evaluation results (10 positive, 10 negative) per requirement. If there are not enough results (e.g., when a requirement is almost always satisfied), we take the maximum number available.

We then have a human annotator manually review the evaluations. For each model output, the annotator compared the predicted label against their own judgment of whether the output satisfied the given requirement. Overall, the evaluators have 95.6% (SD=0.08) agreement rates, indicating a reasonably high level of human–LLM agreement.

A.7 Prompt templates and construction

We use a simple prompt template to construct our experiment prompts. The task descriptions are one-line minimal descriptions as shown in Appendix A.1.

Prompt template for our experiments

[Task description]

Follow the guideline below:
- [requirement 1]
- [requirement 2]
...
- [requirement N]

In our experiments, our goal is to systematically cover requirement subsets, to make sure (a) each prompt has roughly the same complexity in terms of the number of requirements to follow, and (b) different requirements are specified or unspecified the same number of times. We use a simple cyclic design to achieve this, as shown in Figure 11.

A.8 Compute resources used in the experiments

In our first set of experiments (up to Section 3.3), we make 6k inferences with each of the 7 models to obtain model outputs. We then make 840k inferences to evaluate the results with `gpt-4.1-mini`.

In our second experiment (Section 3.5), we make 6k inferences with each of the two models on 3 different prompt configurations (different numbers of requirements). We then make 720k inferences to evaluate the results with `gpt-4.1-mini`.

For prompt optimization experiments (Section 4), we make 16.2k inferences with each prompt optimizer to produce all optimized prompts (324k inferences for evaluation). We then make 6k inferences with 4 different optimization results each (480k inferences for evaluation).

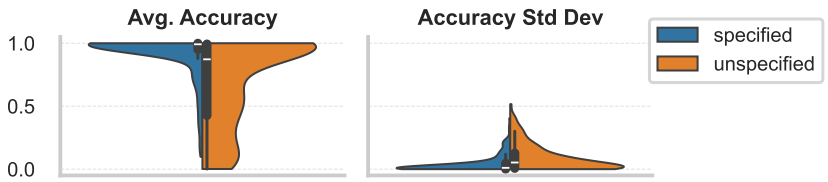

Figure 12: Comparing LLM+Prompts performances on specified requirements vs. unspecified requirements, we found that, overall, LLM+Prompts perform worse and diverge more for unspecified requirements. This is statistically significant even if we consider all other factors and explains a large portion of the variances observed (Table 3).

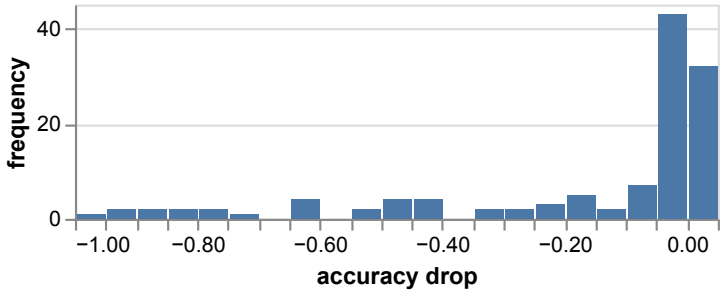

Figure 13: The histogram of average requirement accuracy drops when the prompts include more requirements (N=1→19). We found 37.5% requirements drop significantly by more than 5%.

B Additional experiment results

B.1 ANOVA results

We apply ANOVA to analyze how factors like requirement *scope*, *source*, *category*, or *model* impact LLM+Prompts performance on specified vs. unspecified requirements. For factors that are significant, we use Tukey's HSD test to identify which specific group means are significantly different from each other. Results are reported in Table 3.

Table 3: ANOVA results: We report the F-value and p-value, which quantify the extent to which each variable accounts for the observed variances. We found that whether a requirement is specified has the largest impact on average accuracy (+0.2) and a significant impact on SD (-0.037). Breaking down the requirements, we found models struggle with conditional requirements (-0.09 accuracy, +0.017 SD), but are better at requirements found in existing prompts and format-related requirements.

	Avg. accuracy	**SD of accuracy**	**Acc. delta**	**SD delta**
Specified?	125.87***	48.47***	-	-
Conditional?	19.01***	16.38***	4.55*	3.03
Source	19.99***	36.83***	9.07***	3.63*
Category	23.86***	55.00***	1.01	0.44
Model	1.80	0.57	0.68	1.71

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

B.2 Prompt Optimization Examples

An example of unoptimized prompts

Explain the code snippet.

Follow the guideline below:
- The output should explain the purpose of the code.
- The output should explain scenarios where certain features of the code are particularly beneficial or efficient.
- The output should use analogies and examples to clarify the explanation.
- The output should not describe components or operations not present in the provided code.
- The output should check and explain any potential side effects or state changes that occur during code execution.
- The output should include a precise, step-by-step execution order that aligns with the code.
- The output should break down the code's functionality.
- If technical jargon is used, the output should provide clear explanations for it.
- The output should include information about verifying the setup or configuration before execution, if applicable.
- The output should identify and explain any variables or data structures used in the code snippet.
- The output should explicitly define the scope of explanation without making assumptions about specific use cases.
- The output should aim to make the explanation accessible to someone with minimal coding knowledge.
- If there are error handling mechanisms, the output should accurately describe them and explain how they handle potential errors.
- The output should address potential edge cases tested by the code.
- The output should not exceed 500 words to maintain conciseness and focus.
- The output should provide an example of how at least one function, class, or constant imported from the code can be used.
- The output should detect and describe any dependencies or libraries required by the code snippet.
- The explanation should include potential applications and implications of the coded algorithm.
- The output should provide a high-level summary at the beginning to set the context.

Figure 14: Unoptimized prompts generated from the prompt template (acc=75.4%).

An example of COPRO-optimized prompts

Provide a detailed explanation of the given code snippet, following these guidelines:

1. **Overview**: Begin with a high-level summary that sets the context for the code's purpose and functionality.
2. **Purpose and Benefits**: Clearly explain the code's main objective and highlight scenarios where its features are particularly beneficial or efficient.
3. **Analogies and Examples**: Use relatable analogies and examples to enhance understanding, especially for complex concepts.
4. **Code Breakdown**: Decompose the code into its fundamental components, explaining the role and function of each part.
5. **Step-by-Step Execution**: Offer a precise, sequential walkthrough of how the code executes, ensuring clarity on the process.
6. **State Changes and Side Effects**: Identify any potential side effects or state changes that occur during execution.
7. **Variable and Data Structure Explanation**: Define and explain any variables or data structures used, ensuring comprehension for those with minimal coding knowledge.
8. **Technical Jargon**: Simplify or clarify any technical terms to make the explanation accessible.
9. **Setup Verification**: Include information on verifying the setup or configuration before running the code, if relevant.
10. **Error Handling**: Describe any error handling mechanisms, detailing how they manage potential errors and edge cases.
11. **Dependencies**: Identify any libraries or dependencies required by the code.
12. **Applications and Implications**: Discuss potential applications and implications of the algorithm or functionality provided by the code.
13. **Function Usage Example**: Provide an example of how at least one function, class, or constant from the code can be utilized.
14. **Conciseness**: Ensure the explanation does not exceed 500 words, maintaining focus and clarity.

Figure 15: COPRO-optimized prompts (acc=86.7%). We found COPRO-optimized prompts tend to reorder requirements in a more logical structure, merge related requirements together, and sometimes drop requirements.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The claims are all backed up by our experimental results.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We discuss limitation in our experiment design in Section 3.6.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [NA]

Justification: The paper does not include theoretical results.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: Reproduction steps are described in the code.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: The experiment code and data are all shared.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: All experiment details are described.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: All main results are reported with error bars when applicable.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.

- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We describe compute resources used in the Appendix A.8.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [NA]

    Justification: The paper does not directly have societal impact – it studies behaviors of existing LLMs in a specific setting.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: The paper poses no such risks.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Reused datasets are cited and the licenses are mentioned and respected.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.

- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: Code written and data curated are shared in the paper.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: We describe how we use LLM-assisted requirements curation and LLM-as-a-judge for evaluation in our experiments.

Guidelines:

- The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.