CLARIFYCODEBENCH: EVALUATING CODING AGENTS ON CLARIFYING UNDERSPECIFIED REQUIREMENTS FOR CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-based coding agents have emerged as powerful programming assistants. However, code generation is constrained by requirements that are frequently underspecified in software development: they omit information needed to determine the intended behavior. Although coding agents excel at one-shot code synthesis, their ability to proactively clarify user intent remains underexplored. Code generation benchmarks primarily score final programs, leaving the quality of requirement elicitation less well characterized. We therefore introduce ClarifyCodeBench, an interactive benchmark for evaluating coding agents' capability in resolving requirement underspecification. Constructed from competitive programming tasks, ClarifyCodeBench provides manual annotations of 11 underspecification types, associated clarification questions, and corresponding reference answers. We formalize two metrics to assess interaction quality: Key Question Rate (KQR), which measures the fraction of annotated key questions an agent asks, and Turn-discounted Key Question Rate (TKQR), which additionally rewards earlier coverage. We evaluate six standalone configurations and two configurations of the Claude Code harness. Our evaluation yields three insights: 1) Capability Decoupling: Strong code generation performance does not inherently translate to strong clarification performance; 2) The Reasoning Paradox: Enabling reasoning improves code correctness but reduces clarification coverage; 3) The Multi-underspecification Ceiling: Coding agents' clarification performance degrades sharply as the density of underspecification increases, revealing a bottleneck in resolving multiple missing requirements. Our work emphasizes the need for future coding agents to move from static synthesis to interactive requirement elicitation that establishes the intended behavior of the generated program.

1 INTRODUCTION

LLM-based coding agents are increasingly used as programming assistants (Sergeyuk et al., 2025; Akhoroz & Yildirim, 2025; Paradis et al., 2025; Weisz et al., 2025). Users provide requirements in natural language, and coding agents generate executable code accordingly (Chen et al., 2021; Austin et al., 2021). However, requirements are often underspecified: they omit information needed to determine the intended behavior (Mu et al., 2024), so a coding agent that generates code immediately must take a guess, but a wrong guess would lead to a plausible but incorrect program.

For example, in the task shown in Figure 1, deleting the word "geometric" leaves a task asking whether a sequence is "a progression": an arithmetic and a geometric reading are both plausible, and they disagree already on $A = (1, 2, 3)$. A coding agent that generates code immediately must commit to one reading without revealing its choice, risking a mismatch with the user's intent (Mu et al., 2024; Larbi et al., 2025). A reliable assistant should instead surface the missing information: either by asking the user to resolve it or by stating the assumption under which it proceeds. At the same time, unnecessary questions burden the user. Effective clarification therefore involves more than a single decision about whether to ask. It is a structured elicitation problem: a coding agent must localize each piece of missing information and scale its interaction effort to how much the requirement leaves unsaid. Accordingly, we evaluate the clarification behavior of current coding agents along three dimensions. **Selectivity** asks whether they seek clarification when information

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

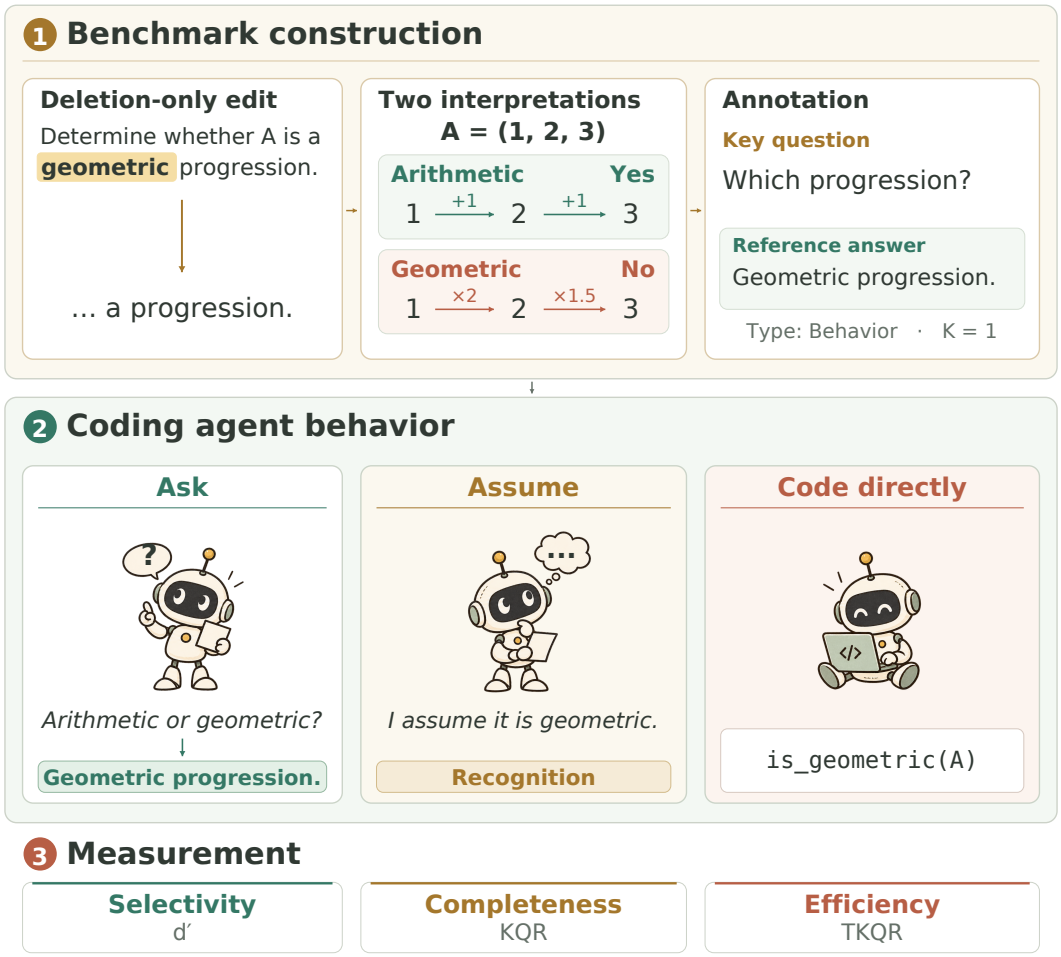


Figure 1: Overview of the construction and interactive evaluation protocol of ClarifyCodeBench.

is missing while avoiding unnecessary questions on complete requirements. **Completeness** asks whether they elicit all specific information that is missing. **Efficiency** asks whether they raise the key questions early and scale their clarification effort to the amount of missing information.

Existing benchmarks, including HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), APPS (Hendrycks et al., 2021), and LiveCodeBench (Jain et al., 2024), mainly evaluate code generation from complete requirements. ConvCodeWorld (Han et al., 2025) studies code generation under feedback, while CodeFlowBench (Wang et al., 2025) studies iterative function reuse across multiple turns. These settings focus on code construction. Evaluating requirement elicitation additionally requires identifying which information an agent requests and how completely it resolves the missing requirements before implementation.

To address this limitation, we introduce ClarifyCodeBench, an interactive benchmark that evaluates how well coding agents clarify underspecified requirements. Each task contains an underspecified requirement, a set of annotated key clarification questions and answers, underspecification types, and an executable test suite for the final implementation. During evaluation, coding agents may ask questions before generating code. Questions targeting annotated omissions receive the corresponding answers. Coding agents may also state explicit assumptions, which provide evidence that they recognized the corresponding missing information but do not constitute clarification. We use competitive programming as a controlled setting for studying clarification. We introduce Key Question Rate (KQR) to measure the fraction of annotated questions an agent asks, and Turn-discounted Key Question Rate (TKQR) to additionally reward earlier questions. We also measure recognition by crediting a question when the agent either asks about it or states a corresponding assumption. Finally, pass@1 on the complete statements measures coding ability separately from clarification.

Building on ClarifyCodeBench, we evaluate six configurations of four LLM backbones and two configurations of the Claude Code harness. Our study leads to three main findings. First, strong code generation performance does not inherently translate to strong clarification performance. Second, enabling reasoning improves code correctness but reduces clarification coverage. Finally, coding agents' clarification performance degrades sharply as the density of underspecification increases, revealing a bottleneck in resolving several missing requirements within the same task.

To this end, this paper makes the following contributions:

- We introduce ClarifyCodeBench, an interactive benchmark that pairs each annotated omission with a key clarification question and a reference answer. Its deletion-only construction provides 11 underspecification types, controlled omission counts, matched complete controls, and executable tests.
- We introduce KQR and TKQR to measure coverage of missing information and interaction efficiency, complementing selectivity and functional correctness.
- We evaluate six standalone configurations and two Claude Code configurations, finding that coding agents struggle to identify and resolve underspecification. Enabling reasoning reduces clarification coverage, and performance drops sharply as underspecification increases.

2 Related Work

2.1 Code Generation Benchmarks

Code generation benchmarks have expanded beyond standalone tasks such as HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and APPS (Hendrycks et al., 2021). LiveCodeBench (Jain et al., 2024) uses recently released competition problems to reduce contamination. ConvCodeWorld (Han et al., 2025) introduces reproducible feedback environments, while CodeFlowBench (Wang et al., 2025) evaluates iterative function reuse across multiple turns. HumanEvalComm (Wu & Fard, 2025) evaluates communication rate, question quality, and functional correctness under ambiguity, inconsistency, and incompleteness. It uses an LLM evaluator to answer questions from the original task statement. ClarifyCodeBench instead supplies fixed reference answers for individually annotated omissions, enabling direct measurement of omission coverage. Table 1 summarizes these evaluation settings. ClarEval (Li et al., 2026) measures intent coverage and dialogue efficiency using predefined intent–response scripts across three ambiguity categories. ClarifyCodeBench uses deletion-only edits, assigns one of 11 types to each omission, and varies the omission count K from one to three. Together with matched complete controls and executable tests, these annotations support joint analysis of selectivity, omission coverage, interaction effort, and functional correctness. Section 5.3 examines how clarification effort changes with K.

2.2 Clarification for Underspecified Requirements

Interactive clarification workflows refine requirements before generation. TiCoder (Fakhoury et al., 2024) helps users formalize intent through tests, and ClarifyGPT (Mu et al., 2024) detects ambiguity, asks clarification questions, and then generates code from the refined requirement. Other work seeks to improve clarification at the model level: ClarifyCoder (Wu et al., 2025) uses synthetic data and fine-tuning to teach models to ask questions rather than guess user intent. These studies develop clarification policies and assess their effect on code generation. Our benchmark complements them by measuring selective questioning, coverage of individual omissions, and the allocation of clarification turns. We use ClarifyGPT as a system-level baseline under the same reference annotations and complete controls.

3 The ClarifyCodeBench Benchmark

3.1 Dataset Construction

Task Construction. We build ClarifyCodeBench on the v6 split of LiveCodeBench (Jain et al., 2024), using its complete statements for controlled editing and its hidden tests for functional scoring.

We create underspecified requirements through **deletion-only** editing: annotators remove information that determines the intended behavior. Edits preserve fluency and the original programming objective. We annotate one to three *requirement omissions* per task, each with a key clarification question and reference answer. Providing all answers restores the complete specification. Two PhD students independently annotate every task and cross-check the edits, questions, and reference answers, resolving disagreements through discussion.

Table 1: Benchmark comparison.

Benchmark	C	I	A	T
HumanEval (2021)	✓	–	–	–
MBPP (2021)	✓	–	–	–
APPS (2021)	✓	–	–	–
LiveCodeBench (2024)	✓	–	–	–
ConvCodeWorld (2025)	✓	✓	–	–
CodeFlowBench (2025)	✓	✓	–	–
HumanEvalComm (2025)	✓	✓	–	△
ClarEval (2026)	✓	✓	✓	△
ClarifyCodeBench	✓	✓	✓	✓

C: functional correctness; I: interaction; A: annotated reference Q&A; T: underspecification types. △: coarse task-level types.

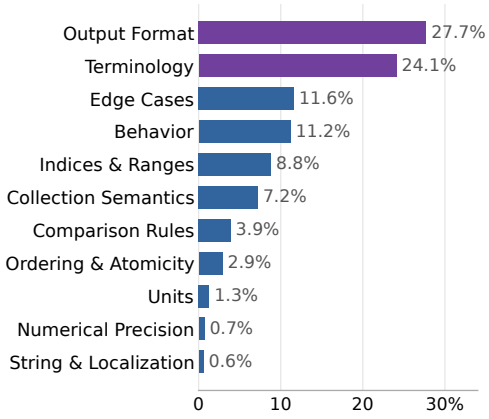


Figure 2: Underspecification type distribution.

Underspecification Types. After task construction, we assign an underspecification type to each omission. The type scheme follows requirement quality principles (ISO/IEC/IEEE, 2018; Davis et al., 1993; Berry et al., 2003) instantiated for code generation. Figure 2 shows the type distribution.

Dataset Summary. The dataset contains 419 underspecified variants from 80 source problems: 199 contain one omission, 169 contain two, and 51 contain three. It supports evaluating clarification and code correctness after receiving answers.

Validation of Annotated Omissions. For each task–agent pair, we sample 3 programs before and 3 after supplying the reference answer. We retain pairs with at least two identical normalized output vectors in each condition. This criterion retains 82.8% of pairs and defines a unique modal behavior for comparison. We compare the two modal output vectors on the source-task test inputs. A difference indicates that the answer changes program behavior. This validation uses no expected outputs and is separate from clarification scoring. Appendix A.3 formalizes this comparison.

3.2 Evaluation Metrics

Terminology. Each agent response is either a *clarification turn*, asking one question, or the final *code generation turn*. Let n denote the number of clarification turns; immediate code generation gives $n = 0$. Let $\mathcal{Q}$ denote a task's annotated key questions and $K = |\mathcal{Q}|$, with $K \in \{1, 2, 3\}$ in ClarifyCodeBench. We set $h_i = 1$ if clarification turn i covers a previously uncovered key question, and $h_i = 0$ otherwise. Thus, repeated questions receive no additional credit.

Key Question Rate. KQR measures the fraction of annotated key questions that the agent asks:

$$\text{KQR} = \frac{1}{K} \sum_{i=1}^{n} h_i. \tag{1}$$

KQR lies in $[0, 1]$. The *full-coverage rate* is the fraction of tasks with all K questions covered (KQR $= 1$). *Recognition* additionally credits matching assumptions through the `[ASSUMPTION]` channel (Section 4.1), using the same judge for both questions and assumptions.

Turn-discounted Key Question Rate. TKQR accounts for when key questions are asked by applying the logarithmic discount used in normalized DCG (Järvelin & Kekäläinen, 2002; 2017):

$$\mathrm{TKQR} = \frac{\mathrm{DCG}}{\mathrm{IDCG}}, \qquad \mathrm{DCG} = \sum_{i=1}^{n} \frac{h_i}{\log_2(i+1)}, \qquad \mathrm{IDCG} = \sum_{i=1}^{K} \frac{1}{\log_2(i+1)}. \tag{2}$$

IDCG corresponds to asking all K key questions in the first K clarification turns. Normalizing over all K questions ensures that both missed and delayed questions reduce the score. TKQR lies in $[0, 1]$ and equals 1 if and only if all K key questions are asked within the first K turns.

Discrimination. To measure selective clarification, we also evaluate each agent configuration on 80 complete control statements ($K = 0$). Worked examples are removed using the same truncation rule as for the underspecified tasks. Any question asked on the control set is treated as a false alarm. Let H be the fraction of underspecified tasks on which the agent asks at least one question, and F the corresponding fraction on the control set. We compute $d' = z(H) - z(F)$, where z is the inverse standard normal CDF (Wichura, 1988). Before applying z, each rate is loglinearly corrected (Hautus, 1995) as $(x + 0.5)/(N + 1)$, where x is the number of positive cases among N tasks. We report d', H, and F using the normalized controls described in Appendix C.1.

4 Evaluation Setup

4.1 Protocol

Interaction. For each task, the agent receives the underspecified requirement and a common system prompt. At each turn, the agent may ask one clarification question or produce its final code, optionally preceded by an explicit assumption about missing information. A clarification question receives the reply described below, after which interaction continues until the agent produces code or reaches the six-turn limit. After the initial prompt, no further instruction directs whether to ask or generate code. Appendix B provides the full system prompt and protocol.

LLM-as-a-Judge. We use an LLM judge (Zheng et al., 2023; Badshah & Sajjad, 2025) to match each clarification question to the annotated key questions. When a match is found, we return the corresponding reference answer to the evaluated agent. Otherwise, the agent receives a fixed neutral reply. This reply provides no information about whether the question targets missing information. Question and assumption matching use GPT-4o (Hurst et al., 2024) with three-sample majority voting.

Coding Agents. We evaluate coding agents built on four LLM backbones under six configurations. Claude Opus 4.8 (Anthropic, 2026b) and GPT-5.5 (OpenAI, 2026a) are each evaluated with and without reasoning. Gemini 3.1 Pro (Google, 2026) and GPT-5 mini (OpenAI, 2026b) are evaluated with reasoning enabled. These six configurations use their provider's default sampling settings. Tables identify agents by backbone and mark reasoning as enabled (✓) or disabled (–). Appendix B.1 gives the reasoning settings and token budgets.

Agent Harness. We additionally evaluate Claude Code (Anthropic, 2026a) with Claude Opus 4.7 and Claude Sonnet 4.6, reporting precision, recall, TKQR, and mean questions per task.

Baseline. We compare the six configurations with ClarifyGPT (Mu et al., 2024). ClarifyGPT samples 25 candidate programs and identifies a task as underspecified when they produce different outputs on the public test inputs. It then generates a batch of clarification questions and regenerates the program using the answers. We adapt its prompts from the released implementation and run it with GPT-5.5 without reasoning. This setting controls for differences in the underlying backbone. Its questions are evaluated against the same annotations as those of the other configurations.

4.2 Functional Correctness

We measure functional correctness using pass@1. A program passes only if its output matches the expected output on every test case. Each final program runs under a 10-second CPU limit and a

2 GiB memory limit; timeouts, crashes, and empty outputs count as failures. For each underspecified variant, we evaluate three conditions while holding the backbone, task, and tests fixed. In DIRECT, the agent generates code from the underspecified requirement without interaction. In INTERACTIVE, the agent follows the protocol above and receives the reference answer when its question matches an annotation. In ORACLE, the agent receives all annotated reference answers before generating code.

We report two paired differences:

$$\Delta_{\text{int}} = \text{Interactive} - \text{Direct}, \qquad \Delta_{\text{oracle}} = \text{Oracle} - \text{Interactive}. \tag{3}$$

The interactive gain measures the improvement obtained through the agent's own clarification. The oracle gain measures the remaining improvement available from the missing information. We also report pass@1 on the complete control statements as a reference measure of coding ability. Appendix C.3 gives the execution procedure and the subsets used for the paired comparisons.

5 RESULTS

5.1 CODING AGENTS SHOW SELECTIVITY IN CLARIFICATION DECISIONS

Table 2: Clarification selectivity and response strategies.

Agent	R	Clarification			Responses (%)	
		H (%)	F (%)	d'	Declared	Silent
Claude Opus 4.8	✓	16.0	**0.0**	1.51	33.4	50.6
	–	55.4	25.0	0.80	10.5	34.1
GPT-5.5	✓	11.9	**0.0**	1.33	19.3	68.7
	–	21.2	**0.0**	**1.71**	3.3	75.4
Gemini 3.1 Pro	✓	47.0	8.8	1.25	14.8	38.2
GPT-5 mini	✓	15.8	10.0	0.25	8.6	75.7

R: reasoning (✓ on; – off). H/F: hit/false-alarm rates. Response rates use all underspecified tasks.

We compare whether each configuration asks at least one question on underspecified requirements and matched complete controls. As shown in Table 2, five of the six configurations ask substantially more often when information is missing, with d' ranging from 0.80 to 1.71. Three configurations never ask on complete controls. GPT-5 mini is the only exception: its d' is 0.25.

These rates describe the decision to ask. Coverage metrics additionally assess whether the questions identify the annotated missing information.

5.2 CODING AGENTS RARELY IDENTIFY THE SPECIFIC MISSING INFORMATION

Clarification selectivity does not translate into coverage of the specific missing information. Table 3 shows that KQR remains low across all configurations. Claude Opus 4.8 without reasoning achieves the highest KQR of 0.230, while four of the six configurations remain below 0.11. Including explicit assumptions raises the highest recognition score to 0.281. Thus, at least 71.9% of the annotated missing information is neither queried nor acknowledged through an explicit assumption. Therefore, distinguishing when clarification is needed does not imply an ability to identify what information is missing.

Table 3: Clarification coverage and interaction effort.

Agent	Reasoning	Information coverage			Full coverage (%)			Turns
		KQR	TKQR	Recog.	$K=1$	$K=2$	$K=3$	
Claude Opus 4.8	✓	0.076	0.086	0.144	7.0	1.2	0.0	0.17
	–	**0.230**	**0.251**	**0.281**	**25.6**	**4.1**	**2.0**	0.74
GPT-5.5	✓	0.056	0.063	0.141	5.5	0.0	0.0	0.13
	–	0.101	0.112	0.130	10.6	1.8	0.0	0.23
Gemini 3.1 Pro	✓	0.187	0.208	0.243	20.6	1.8	0.0	0.53
GPT-5 mini	✓	0.066	0.072	0.097	7.5	1.2	0.0	0.22

Recog.: recognition, including declared assumptions. Full coverage: tasks with all K key questions asked. Turns: mean clarification turns. Bold indicates the highest coverage.

Table 4: Clarification with Claude Code.

Backbone	Precision	Recall	TKQR	Questions
Opus 4.7	0.568	0.220	0.255	0.55
Sonnet 4.6	0.383	0.078	0.081	0.29

Questions: mean number asked per task.

The Claude Code results in Table 4 show a similar coverage bottleneck. Opus 4.7 reaches higher precision, recall, and TKQR than Sonnet 4.6, but recall remains 0.220. Both average fewer than one question per task. These harness configurations therefore still leave a substantial share of the missing information unresolved.

5.3 Clarification Effort Does Not Scale with Missing Information

Low coverage may reflect late questioning or premature termination. The results support the second explanation. TKQR remains close to KQR across configurations: matched questions are generally raised early. The primary failure is insufficient questioning. This limitation becomes more pronounced as the amount of missing information increases. Table 3 shows that the best full-coverage rate falls from 25.6% at $K = 1$ to 4.1% at $K = 2$. At $K = 3$, five configurations never achieve full coverage, whereas the remaining configuration reaches only 2.0%. The six-turn limit does not account for this decline: agents use only 0.13 to 0.74 clarification turns on average. The six-turn limit is reached only twice across all 2,514 recorded dialogues.

Table 5 shows that every configuration asks more questions as K increases. However, from $K = 1$ to $K = 3$, the number of questions required grows by two, while mean turns grow by only 0.18–0.43. The most inquisitive configuration averages 1.04 questions per task at $K = 3$.

Turns per omission therefore fall or stagnate as K rises for every configuration except GPT-5.5 with reasoning, whose rate rises only from 0.07 to 0.10. Agents ask their few key questions early, but clarification effort remains nearly fixed as missing information grows.

Table 5: Clarification effort by K.

Agent	R	$K = 1$	$K = 2$	$K = 3$	Δ
Claude Opus 4.8	✓	0.12	0.21	0.29	+0.18
	–	0.61	0.81	1.04	+0.43
GPT-5.5	✓	0.07	0.14	0.31	+0.24
	–	0.14	0.28	0.41	+0.28
Gemini 3.1 Pro	✓	0.42	0.59	0.80	+0.38
GPT-5 mini	✓	0.16	0.27	0.33	+0.18

Δ: change from $K = 1$ to $K = 3$.

5.4 Strong Coding Ability Does Not Imply Effective Clarification

Table 6: Coding and clarification.

Agent	Reasoning	pass@1 (%)	KQR
Claude Opus 4.8	✓	81.2	0.076
	–	62.5	**0.230**
GPT-5.5	✓	**86.2**	0.056
	–	55.0	0.101
Gemini 3.1 Pro	✓	81.2	0.187
GPT-5 mini	✓	66.2	0.066

pass@1 uses complete control statements.
Bold values denote the best result per metric.

Strong code generation performance does not imply effective clarification ability. Pass@1 on complete controls measures coding ability without requiring agents to resolve underspecification. GPT-5.5 with reasoning achieves the highest pass@1 of 86.2% but the lowest KQR of 0.056. In contrast, Claude Opus 4.8 without reasoning achieves the highest KQR of 0.230 but only 62.5% pass@1. Therefore, coding and clarification performance follow different rankings.

This distinction also affects functional correctness. Effective clarification should recover a substantial share of the improvement available when all missing information is provided. Table 7 shows that INTERACTIVE changes pass@1 by only -0.5 to $+4.8$ percentage points relative to DIRECT. In contrast, ORACLE provides all reference answers and improves pass@1 by a further 3.3 to 10.3 points over INTERACTIVE. This additional improvement is positive across all six evaluated agent configurations.

5.5 Reasoning Improves Coding but Reduces Clarification Coverage

Table 6 shows a consistent tradeoff with reasoning enabled. For Claude Opus 4.8, enabling reasoning raises pass@1 from 62.5% to 81.2%, while KQR falls from 0.230 to 0.076. GPT-5.5 shows the same pattern: pass@1 increases from 55.0% to 86.2%, while KQR decreases from 0.101 to 0.056. Reasoning therefore strengthens code generation while reducing clarification coverage.

Table 7: Effect of clarification on functional correctness.

Agent	Reasoning	pass@1 (%)			Change (pp)	
		Direct	Interactive	Oracle	Δ_{int}	Δ_{oracle}
Claude Opus 4.8	✓	67.6	71.5	76.7	+3.9	+5.2
	–	56.6	57.0	63.5	+0.5	+6.4
GPT-5.5	✓	**75.7**	**75.2**	**82.1**	−0.5	+6.9
	–	48.2	48.7	52.0	+0.5	+3.3
Gemini 3.1 Pro	✓	66.6	68.0	76.1	+1.4	+8.1
GPT-5 mini	✓	51.1	55.8	66.1	**+4.8**	+10.3

Δ_{int}: Interactive − Direct; Δ_{oracle}: Oracle − Interactive. Bold: best pass@1 and interactive gain.

The response distributions in Table 2 explain this tradeoff. For Claude Opus 4.8, reasoning reduces the ask rate from 55.4% to 16.0% and increases the assumption declaration rate from 10.5% to 33.4%. For GPT-5.5, the ask rate decreases from 21.2% to 11.9%, while the declaration rate rises from 3.3% to 19.3%. These assumptions do not compensate for the missing questions. Recognition changes only from 0.130 to 0.141 for GPT-5.5 and falls from 0.281 to 0.144 for Claude Opus 4.8. Therefore, reasoning shifts responses from questions that users can answer toward assumptions made without user input, without consistently improving recognition of the missing information.

5.6 Coding Agents Prioritize Core Behavior over Boundary Conditions

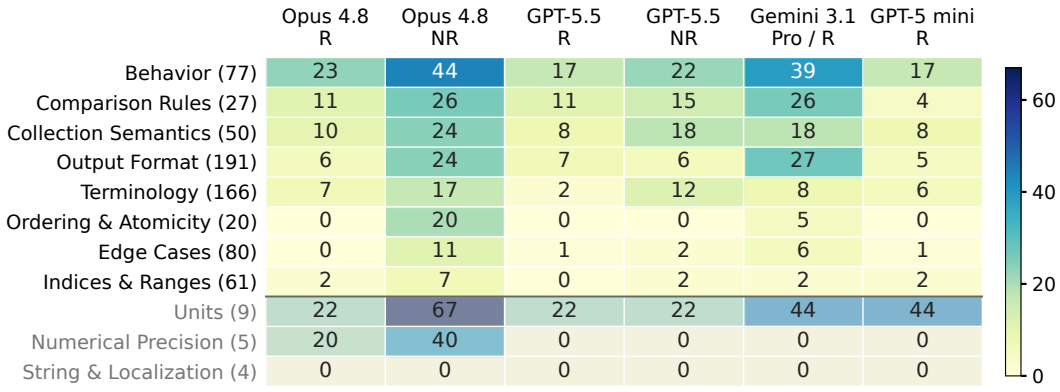


Figure 3: Question hit rate (%) by underspecification type. R/NR: reasoning on/off. Parentheses give counts; shaded types have fewer than ten omissions and are excluded from the analysis.

Figure 3 shows that, among the eight types with at least ten annotated omissions, the hit rate ranges from 27.1% for Behavior to 2.2% for Indices & Ranges. This ordering is stable across configurations, with a median pairwise rank correlation of 0.83. Every configuration ranks Behavior first, while Edge Cases, Indices & Ranges, and Ordering & Atomicity consistently receive the lowest coverage. This shared ordering reveals a systematic preference for core behavior over boundary conditions. Coding agents are more likely to ask what a program should compute than under which conditions the computation must remain correct. They often capture the central operation while overlooking edge cases, index ranges, and ordering requirements. Consequently, an implementation may appear plausible on typical inputs yet remain incorrect at critical boundaries.

5.7 ClarifyGPT Improves Coverage at the Cost of Selectivity

Table 8 compares ClarifyGPT with its GPT-5.5 backbone without reasoning. ClarifyGPT raises KQR from 0.101 to 0.288, surpassing the best standalone configuration at 0.230. This improvement accompanies an increase from 0.23 to 3.74 questions per task. ClarifyGPT also achieves 11.8% full coverage at $K = 3$, compared with 2.0% for the best standalone configuration.

ClarifyGPT asks all questions in one batch. To compute TKQR, we assign matched questions to the earliest clarification turns. All configurations use the same complete controls to measure false alarms.

Table 8: ClarifyGPT comparison with GPT-5.5 without reasoning.

Selectivity and interaction					Information coverage					
Agent	H (%)	F (%)	d'	Questions	Agent	KQR	TKQR	$K=1$	$K=2$	$K=3$
GPT-5.5	21.2	**0.0**	**+1.71**	0.23	GPT-5.5	0.101	0.112	10.6	1.8	0.0
ClarifyGPT	54.7	50.0	+0.12	3.74	ClarifyGPT	**0.288**	**0.304**	**28.6**	**17.2**	**11.8**

$K = 1, 2, 3$: full coverage (%). Questions: mean clarification questions per task.

The higher coverage does not reflect better identification of when clarification is needed. ClarifyGPT asks on 54.7% of underspecified requirements and 50.0% of complete controls, yielding $d' = 0.12$. In contrast, the same backbone under our interaction protocol achieves $d' = 1.71$ and never asks on a complete control. Therefore, ClarifyGPT gains coverage by asking broadly across both underspecified and complete requirements rather than by asking selectively. No evaluated system combines high coverage of missing information with selective clarification decisions.

6 Discussion

6.1 Effective Clarification Requires Selectivity and Coverage

Our results distinguish deciding whether clarification is needed from identifying what information must be clarified. Reasoning shifts responses from questions toward unverified assumptions, whereas ClarifyGPT increases coverage by asking broadly. Neither achieves both selective interaction and broad coverage. The additional gains under Oracle show that the unaddressed information can materially affect the functional correctness of the generated program.

Explicit assumptions expose an agent's interpretation, while clarification lets the user resolve that choice. The reasoning results show that stating assumptions does not consistently improve recognition. The decline in full coverage at larger K connects interaction effort to the number of unresolved decisions, motivating a clarification policy that follows the task's information needs before generating code that implements the user's intended behavior.

Coding agents need to track unresolved information, prioritize information that can alter program behavior, and adjust interaction effort to the amount of missing information. Evaluation should separately measure clarification selectivity, coverage, and interaction efficiency. Neither functional correctness nor asking rate alone reveals which stage of clarification fails.

6.2 Agreement across LLM Judges

The coverage metrics use GPT-4o to match questions to annotations with a fixed prompt and three-sample majority voting. We assess agreement with Claude Opus 4.8 and Gemini 3.1 Pro by rescoring every match. Pairwise agreement on omission coverage ranges from 0.93 to 0.95, while Cohen's κ (Cohen, 1960) ranges from 0.82 to 0.87. All three judges agree on 91% of omissions, and the pooled coverage estimated by each alternative judge differs from that of GPT-4o by less than 0.02. Recognition, including explicit assumptions, yields Cohen's κ of 0.67–0.72.

7 Conclusion

We introduced ClarifyCodeBench, an interactive benchmark that directly measures how well coding agents clarify underspecified requirements before generating code. Annotated omissions, key questions, and reference answers separate clarification ability from functional correctness. Our results distinguish reacting to underspecification from resolving it. Agents often distinguish whether clarification is needed, but fail to elicit most of the missing information. Clarification ability does not follow coding strength. Reasoning shifts interaction toward agent assumptions, while clarification effort changes little as more information is omitted. A dedicated clarification pipeline improves coverage, but shows that asking more questions alone cannot provide both complete and selective clarification. Future work should extend this evaluation to naturally underspecified requirements and real user intent. Ultimately, reliable coding agents must identify when requirements do not yet provide enough information to determine the intended program.

AI USE STATEMENT

Generative AI tools were used to polish the language, improve the readability of the manuscript, and create the decorative robot illustrations in Figure 1. The authors are responsible for the final content.

REPRODUCIBILITY STATEMENT

Section 3 describes benchmark construction, annotation, and validation and defines the evaluation metrics, while Section 4 specifies the evaluation protocol. Appendix A provides the taxonomy and formalizes annotation validation. Appendix B provides the shared prompt, token budgets, and reasoning configurations. Appendix C documents normalization, type attribution, and functional correctness scoring, and Appendix D describes the ClarifyGPT implementation.

REFERENCES

Mehmet Akhoroz and Caglar Yildirim. Conversational AI as a coding assistant: Understanding programmers' interactions with and expectations from large language models for coding. *arXiv preprint arXiv:2503.16508*, 2025.

Anthropic. Claude Code. `https://code.claude.com/docs/en/overview`, 2026a. Accessed September 26, 2026.

Anthropic. Introducing Claude Opus 4.8. `https://www.anthropic.com/news/claude-opus-4-8`, 2026b. Accessed September 26, 2026.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Sher Badshah and Hassan Sajjad. Reference-guided verdict: LLMs-as-judges in automatic evaluation of free-form QA. In *Proceedings of the 9th Widening NLP Workshop*, pp. 251–267. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.winlp-main.37. URL `https://aclanthology.org/2025.winlp-main.37/`.

Daniel M. Berry, Erik Kamsties, and Michael M. Krieger. From contract drafting to software specification: Linguistic sources of ambiguity, 2003. URL `https://cs.uwaterloo.ca/~dberry/handbook/ambiguityHandbook.pdf`. A handbook, version 1.0.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960. doi: 10.1177/001316446002000104.

Alan Davis, Scott Overmyer, Kathleen Jordan, Joseph Caruso, Fatma Dandashi, Anhtuan Dinh, Gary Kincaid, Glen Ledeboer, Patricia Reynolds, Pradip Sitaram, et al. Identifying and measuring quality in a software requirements specification. In *[1993] Proceedings First International Software Metrics Symposium*, pp. 141–152. IEEE, 1993.

Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K Lahiri. LLM-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Transactions on Software Engineering*, 50(9):2254–2268, 2024.

Google. Gemini 3.1 Pro. `https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-1-pro/`, 2026. Accessed September 26, 2026.

Hojae Han, Seung-won Hwang, Rajhans Samdani, and Yuxiong He. ConvCodeWorld: Benchmarking conversational code generation in reproducible feedback environments. *arXiv preprint arXiv:2502.19852*, 2025.

Michael J. Hautus. Corrections for extreme proportions and their biasing effects on estimated values of d'. *Behavior Research Methods, Instruments, & Computers*, 27(1):46–51, 1995. doi: 10.3758/BF03203619.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with APPS. *arXiv preprint arXiv:2105.09938*, 2021.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. GPT-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

ISO/IEC/IEEE. ISO/IEC/IEEE 29148:2018: Systems and software engineering—life cycle processes—requirements engineering. Technical report, International Organization for Standardization, 2018. URL https://www.iso.org/standard/72089.html.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002. doi: 10.1145/582415.582418.

Kalervo Järvelin and Jaana Kekäläinen. IR evaluation methods for retrieving highly relevant documents. *ACM SIGIR Forum*, 51(2):243–250, 2017.

Maya Larbi, Amal Akli, Mike Papadakis, Rihab Bouyousfi, Maxime Cordy, Federica Sarro, and Yves Le Traon. When prompts go wrong: Evaluating code model robustness to ambiguous, contradictory, and incomplete task descriptions. *arXiv preprint arXiv:2507.20439*, 2025.

Jialin Li, Yuan Wu, and Yi Chang. ClarEval: A benchmark for evaluating clarification skills of code agents under ambiguous instructions. *arXiv preprint arXiv:2603.00187*, 2026.

Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. ClarifyGPT: A framework for enhancing LLM-based code generation via requirements clarification. *Proceedings of the ACM on Software Engineering*, 1(FSE):2332–2354, 2024.

OpenAI. Introducing GPT-5.5. https://openai.com/index/introducing-gpt-5-5/, 2026a. Accessed September 26, 2026.

OpenAI. GPT-5 mini: Model documentation. https://developers.openai.com/api/docs/models/gpt-5-mini, 2026b. Accessed September 26, 2026.

Elise Paradis, Kate Grey, Quinn Madison, Daye Nam, Andrew Macvean, Vahid Meimand, Nan Zhang, Benjamin Ferrari-Church, and Satish Chandra. How much does AI impact development speed? An enterprise-based randomized controlled trial. In *SEIP@ICSE*, pp. 618–629. IEEE, 2025.

Agnia Sergeyuk, Yaroslav Golubev, Timofey Bryksin, and Iftekhar Ahmed. Using AI-based coding assistants in practice: State of affairs, perceptions, and ways forward. *Information and Software Technology*, 178:107610, 2025. doi: 10.1016/j.infsof.2024.107610.

Sizhe Wang, Zhengren Wang, Dongsheng Ma, Yongan Yu, Rui Ling, Zhiyu Li, Feiyu Xiong, and Wentao Zhang. CodeFlowBench: A multi-turn, iterative benchmark for complex code generation. *arXiv preprint arXiv:2504.21751*, 2025.

Justin D. Weisz, Shraddha Kumar, Michael Muller, Karen-Ellen Browne, Arielle Goldberg, Ellice Heintze, and Shagun Bajpai. Examining the use and impact of an AI code assistant on developer productivity and experience in the enterprise. In *CHI Extended Abstracts*, pp. 673:1–673:13. ACM, 2025. doi: 10.1145/3706599.3706670.

Michael J Wichura. Algorithm AS 241: The percentage points of the normal distribution. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 37(3):477–484, 1988. URL https://www.jstor.org/stable/2347330.

Jie JW Wu and Fatemeh H Fard. HumanEvalComm: Benchmarking the communication competence of code generation for LLMs and LLM agents. *ACM Transactions on Software Engineering and Methodology*, 34(7):1–42, 2025. doi: 10.1145/3715109.

Jie JW Wu, Manav Chaudhary, Davit Abrahamyan, Arhaan Khaku, Anjiang Wei, and Fatemeh H. Fard. Can code language models learn clarification-seeking behaviors? *arXiv preprint arXiv:2504.16331*, 2025.

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.

A Benchmark Construction

A.1 Source Tasks and Annotation

LiveCodeBench v6 supplies source statements, worked examples, and hidden tests. Annotators remove one to three details while preserving the programming objective and record a key question and reference answer for each omission.

Section 3 describes the independent annotation, cross-checking, and behavioral validation procedures.

A.2 Underspecification Taxonomy

Table 9 defines the eleven underspecification types and gives an example question for each. Figure 2 reports their distribution across annotated omissions.

Table 9: Taxonomy of underspecified coding requirements.

Type	Definition	Example question
Terminology	A domain term, action, or state is undefined, overloaded, or open to multiple interpretations.	What does each of the symbols `#`, `.`, and `?` in the grid represent?
Behavior	The required function, objective, or side effect is underspecified, so the intended behavior is not uniquely determined.	Should this operation modify the original object or return a new one?
Edge Cases	Boundary or exceptional conditions are not specified, leaving behavior unclear for special inputs.	What should be returned when no valid solution exists?
Indices & Ranges	Index bases, interval boundaries, or inclusion rules are underspecified.	Are the endpoints inclusive, and are indices 0-based or 1-based?
Ordering & Atomicity	Temporal order, simultaneity, or indivisible execution assumptions are unclear.	Are these updates processed one by one or applied atomically?
Output Format	The required output structure, layout, or presentation rule is missing or unclear.	Should each answer be printed on a separate line?
Comparison Rules	The comparison key, tie-breaking rule, or stability requirement is not specified.	How should ties be broken when two items have the same score?
Units	A quantity is specified without a clear unit, scale, prefix, or dimensional convention.	Should the result be reported in seconds, minutes, or another unit?
Collection Semantics	A collection, container, or state object is mentioned, but its membership, update rule, or access semantics are underspecified.	What makes an element active, and how should it be retrieved?
Numerical Precision	Precision, rounding, tolerance, or error handling requirements are unclear.	How many decimal places should be printed?
String & Localization	String handling conventions, such as case sensitivity or the accepted spellings of an output token, are not specified.	Is the output comparison case-insensitive, so that `YES` and `yes` are both accepted?

A.3 Validation of Annotated Omissions

Following the sampling and consistency criterion in Section 3, we compare modal program behavior before and after supplying the reference answer.

Let π_A and π_G be programs with the modal output vectors before and after supplying the answer. For task t and agent m, we define the behavioral relevance of an omission as

$$\begin{aligned} \mathrm{DEC}(t, m) &= \mathbf{1}[\phi(\pi_A) \neq \phi(\pi_G)]\,, \\ \phi(\pi) &= (\mathrm{canon}(\pi(x)))_{x \in I(t)}\,. \end{aligned} \tag{4}$$

Here $I(t)$ contains the source-task test inputs, and canon normalizes program outputs. A nonzero value indicates that supplying the answer changes program behavior.

B Interaction Protocol

Each interaction permits at most six turns. The agent asks one question or submits code, optionally preceded by `[ASSUMPTION]` blocks. GPT-4o matches questions by three-sample majority voting and returns the reference answer or a fixed neutral reply. All configurations share the prompt in Figure 5, which demonstrates code generation, clarification, and assumption declaration.

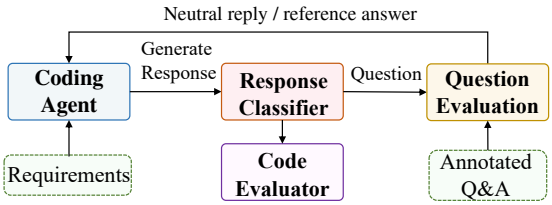


Figure 4: Interactive evaluation procedure.

B.1 Run Configurations

We use provider-default sampling. Completion budgets are $8{,}192$ tokens for GPT-5 mini and Claude Opus 4.8 without reasoning, $32{,}768$ for Gemini 3.1 Pro and both GPT-5.5 configurations, and $64{,}000$ for Claude Opus 4.8 with reasoning. Limits include reasoning where applicable.

The GPT-5.5 ablation changes the reasoning parameter at a fixed endpoint and token budget. The Claude Opus 4.8 configuration without reasoning uses the provider-supported non-thinking setting.

C Measurement Details

C.1 Control Specification Normalization

Complete controls and underspecified variants both exclude worked examples. We normalize the controls by collapsing blank lines while preserving the specification content. All reported hit rates, false-alarm rates, and d' values use these normalized controls.

C.2 Coverage by Underspecification Type

The 199 single-omission tasks each contain one key question and one type, defining the question-level mapping for all variants. We canonicalize minor wording variations before aggregation and count each omission separately, including those sharing a type within one task. Type-aggregated hits reproduce each configuration's overall coverage of annotated omissions.

C.3 Functional Correctness

We extract each final program with its entry-point guard preserved and apply the execution limits in Section 4.2. Passing requires standard output to match every expected output, compared line by line after stripping trailing whitespace and dropping trailing blank lines. The 80 source problems contain $3{,}224$ tests, with 2–45 per problem and nonempty expected outputs.

The same executor scores the complete controls (Table 6) and all three conditions in Table 7. Paired comparisons use variants completed in every condition: 419 for five configurations and 386 for Claude Opus 4.8 with reasoning. Scoring makes no additional model calls.

You are a professional software developer. Your primary goal is to write Python code based on user requirements. *System Message*

Before you start coding, you must perform a critical first step: Carefully evaluate the requirements. *Task Policy*

1. If requirements are clear: Write the code directly.
2. If requirements are unclear: Do not guess. Ask a clarifying question. You may ask as many clarifying questions as you need, one per turn, and you will receive an answer to each before deciding again.
3. If you write code even though the requirement leaves some constraint unstated, you may first declare each constraint you are guessing at, one per `[ASSUMPTION]` block. Declaring assumptions is optional, is neither rewarded nor penalized, and does not affect how your code is evaluated.

Your response must be in one of the following two formats, with no extra text. *Output Format*

```
[ASSUMPTION]
{A constraint you are guessing at.  Repeat this block
for each one, or omit it entirely.}
[/ASSUMPTION]
[CODE]
{Your code here}
[/CODE]
```

or

```
[QUESTION]
{Your single question here}
[/QUESTION]
```

Few-shot Demonstration

```
[EXAMPLE1]
...
[CODE] ...  [/CODE]
[/EXAMPLE1]
[EXAMPLE2]
...
[QUESTION] ...  [/QUESTION]
[/EXAMPLE2]
[EXAMPLE3]
...
[ASSUMPTION] ...  [/ASSUMPTION]
[CODE] ...  [/CODE]
[/EXAMPLE3]
```

Figure 5: System prompt for the agent's clarification policy. The agent asks one question per turn or submits code, optionally preceded by assumption declarations. The three demonstrations are abridged to their channel tags.

D ClarifyGPT Implementation

ClarifyGPT (Mu et al., 2024) groups sampled programs by their output vectors; two or more groups trigger clarification and code regeneration. We use the implementation at commit `543b34b`, keep the question-generation prompt unchanged, and adapt the code-generation instruction from function signatures to stdin/stdout programs for the benchmark tasks.

We sample 25 programs per task at temperature 0.8 and top-p 0.95, using GPT-5.5 without reasoning. Clustering uses normalized standard output, with crashes and timeouts treated as distinct outcomes. Each cluster contributes one representative to the question prompt. The detector uses at most 30 source-task test inputs, without their expected outputs. Questions receive answers through the same GPT-4o judge and reference annotations used for the other agents. ClarifyGPT generates a batch of questions in one turn; for TKQR, we assign matched questions to the earliest clarification turns.