CONTRA: DISCOVERING AND QUALIFYING BEHAVIOR-CHANGING QUESTIONS FOR SELECTIVE CLARIFICATION IN LLM CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents can generate code that appears correct but implements behavior the user never intended. This mismatch can arise when an agent silently resolves underspecified requirements through its own assumptions. As subsequent development builds on these assumptions, correcting the resulting behavior can become increasingly costly. Early clarification can help prevent such mismatches, but unnecessary questions can interrupt developers and slow down development. Existing methods struggle to identify key clarification questions while avoiding unnecessary ones. Therefore, we propose CONTRA, a training-free method that combines broad question discovery with semantic and execution-based question qualification. CONTRA first generates candidate questions and filters out those unrelated to required behavior or already resolved by the requirement. For each remaining question, it generates programs conditioned on two plausible answers and checks for stable behavioral differences on shared inputs. It then uses the interaction history to select among qualified questions or stop asking. Experiments on ClarifyCodeBench show that CONTRA achieves the highest F1 with all four coding agents, exceeding the best baseline macro-average F1 by 13.88 percentage points. With the same LLM and evaluation protocol, CONTRA also achieves higher clarification recall and F1 than the coding harnesses Claude Code and OpenHands. To support practical use, we also implement CONTRA as a Claude Code plugin that integrates selective clarification into everyday development.

1 INTRODUCTION

Coding agents are increasingly used to generate and modify code from natural-language requirements (Hong et al., 2024; Wang et al., 2025; Yang et al., 2024). However, these requirements often leave important aspects of program behavior underspecified, such as how to handle edge cases or format outputs (Fakhoury et al., 2024; Mu et al., 2024; Fang et al., 2026). Faced with such requirements, an agent may silently adopt one plausible interpretation and proceed with implementation. The resulting code may be correct under that interpretation, yet fail to match the user's intended behavior (Fakhoury et al., 2024; Vijayvargiya et al., 2026b; Edwards & Schuster, 2026). If this mismatch goes unnoticed, subsequent development may build on the same assumptions, making correction increasingly costly. Therefore, clarification helps make these assumptions explicit and resolve the underlying choices with the user before the code becomes the basis for further development.

However, excessive questioning can impose unnecessary interaction costs on developers (Zhang & Choi, 2025; Vijayvargiya et al., 2026a). For example, an agent may ask questions already answered by the requirements or earlier interactions, or raise questions unrelated to the task. This motivates selective clarification, where an agent seeks additional user input only when needed (Kuhn et al., 2022). For coding agents, this requires identifying necessary questions within a task while avoiding redundant or irrelevant ones.

Our results in Table 1 illustrate this trade-off. Direct Prompting (Vijayvargiya et al., 2026b; Fang et al., 2026) and Ask-or-Assume (Edwards & Schuster, 2026) rely on the model to decide whether to ask, but often miss necessary questions. By comparison, ClarifyGPT (Mu et al., 2024) generates questions from behavioral disagreement among sampled programs, but asks many unnecessary questions. Such

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.

disagreement can reflect implementation errors even when the requirements are clear. Therefore, selective clarification requires both broad question discovery and careful qualification.

To address this challenge, we propose CONTRA, which combines broad question discovery with careful qualification. It first discovers candidate questions and checks whether each is relevant to the requirement and remains unresolved. For each remaining question, CONTRA generates programs conditioned on two plausible answers and checks for stable behavioral differences on shared inputs. During interaction, it uses the user's answers to select the next qualified question or stop asking. This design identifies key clarification questions while filtering out unnecessary ones.

We evaluate CONTRA on ClarifyCodeBench (Fang et al., 2026) with coding agents built on GPT-5-mini (OpenAI, 2025), GPT-5.5 (OpenAI, 2026), Grok 4.5 (SpaceXAI, 2026), and Gemini 3.1 Pro (Google DeepMind, 2026). Under the same five-question budget, we compare CONTRA with Direct Prompting, Ask-or-Assume (Edwards & Schuster, 2026), and ClarifyGPT (Mu et al., 2024). CONTRA achieves the highest F1 and turn-discounted key question rate (TKQR) for all four agents. Its macro-average F1 and TKQR reach 41.20% and 38.32%, exceeding the best baseline for each metric by 13.88 and 12.33 percentage points, respectively. Furthermore, using Qwen3.8-27B (Qwen Team, 2026) and the same evaluation protocol, CONTRA achieves higher recall and F1 than Claude Code (Anthropic, 2025) and OpenHands (Wang et al., 2025). Its F1 reaches 19.86%, compared with 9.63% for Claude Code and 10.61% for OpenHands. Finally, we integrate CONTRA into Claude Code as a plugin that identifies and reports implementation decisions requiring user clarification.

Our main contributions are as follows.

- We propose CONTRA, a selective clarification method that combines broad question discovery, semantic and execution-based question qualification, and adaptive question selection. Its execution-based qualification links candidate questions to behavioral differences between programs generated under alternative answers.
- We demonstrate consistent improvements on ClarifyCodeBench across four coding agents, achieving the highest F1 and TKQR among the compared methods. Under the same LLM and evaluation protocol, CONTRA also achieves higher clarification recall and F1 than the coding harnesses Claude Code and OpenHands.
- We implement CONTRA as a Claude Code plugin that identifies and reports implementation decisions requiring clarification, helping developers confirm intended program behavior during everyday development.

2 RELATED WORK

Requirement clarification and specification refinement. Prior work improves LLM code generation by making user intent more explicit. CodeClarQA (Li et al., 2023) provides questions and answers that enrich underspecified requirements for code generation. TiCoder (Fakhoury et al., 2024) elicits user feedback on generated tests and uses that feedback to formalize intent and rank candidate programs. ClarifyCoder (Wu et al., 2025) uses synthetic data and instruction tuning to teach code LLMs to recognize underspecification and request clarification. Other methods refine specifications without user interaction. SpecFix (Jia et al., 2025) analyzes the behavioral distribution of sampled programs and maps changes in that distribution back to a repaired problem description. Specine (Tian & Chen, 2026) identifies mismatches between an input specification and the specification inferred from generated code, then constructs an aligned specification for subsequent generation. While these methods refine requirements, CONTRA qualifies individual questions by checking relevance, whether they remain unresolved, and whether plausible answers change program behavior.

Selective clarification. Selective clarification concerns deciding when additional user input is necessary. CLAM (Kuhn et al., 2022) studies whether an LLM should answer directly or request clarification, while IntentSim (Zhang & Choi, 2025) estimates clarification utility from uncertainty over simulated user intents. Learning-based approaches use simulated future turns to label preferences (Zhang et al., 2025), or train clarification policies using task relevance and user answerability (Vijayvargiya et al., 2026a). In code generation, ClarifyGPT (Mu et al., 2024) uses behavioral disagreement among programs sampled from the same requirement to decide whether to clarify. Ask-or-Assume (Edwards & Schuster, 2026) uses the coding agent or a separate intent agent to decide

when user input is needed. CONTRA evaluates each candidate question through semantic checks and behavioral verification, then uses the interaction history to select qualified questions that still require user input.

3 METHODOLOGY

3.1 PROBLEM FORMULATION

Let S denote a natural-language programming requirement. If S is underspecified, a clarification question q requests the additional user input needed to implement the intended behavior.

Given S and a question budget B, the agent may ask at most B questions. At step t, the interaction history contains all previous questions and user answers,

$$H_t = \big[(q_1, a_1), \ldots, (q_{t-1}, a_{t-1})\big],$$

where a_i is the user's answer to question q_i. Based on S and H_t, the agent either asks another question or stops. The objective is to identify and ask key clarification questions while avoiding unnecessary ones, including questions unrelated to the task or already answered by S or H_t.

3.2 OVERVIEW

Figure 1 shows the three stages of CONTRA. Question Discovery constructs a broad candidate set Q_{gen}. Question Qualification applies semantic checks and behavioral verification to obtain Q_{qual}. Finally, Adaptive Question Selection uses the history to select a qualified question or stop.

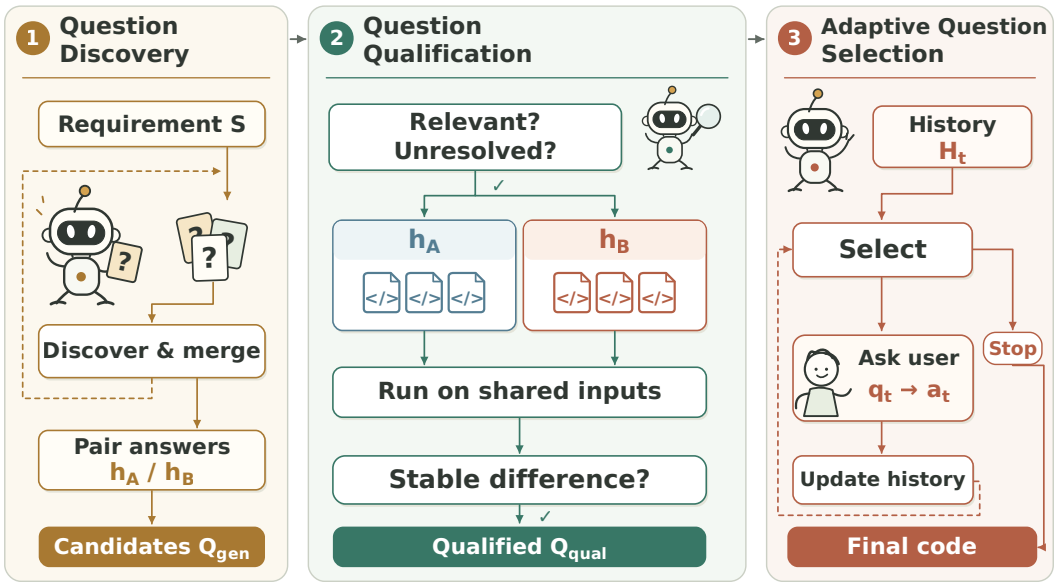

Figure 1: Overview of CONTRA for selective clarification.

3.3 QUESTION DISCOVERY

Question Discovery generates a broad set of candidate clarification questions from S. Each candidate includes a question and two plausible answers, which support candidate review and merging.

To expand the candidate set, each discovery round combines context-aware and independent generation. Context-aware generation uses S and the current candidate set to search for additional questions, while independent generation uses only S. The new and existing candidates are jointly reviewed to remove duplicates and merge overlapping questions. Discovery stops after two consecutive rounds add no distinct questions or a predefined round limit is reached. After discovery, CONTRA regenerates

two plausible answers for each retained question q as requirement completions h_A and h_B. The completions should be compatible with S and differ only in their answer to q. This constraint links subsequent behavioral differences to the question being evaluated and avoids introducing unrelated assumptions. Questions with completion pairs form Q_{gen} for Question Qualification.

3.4 Question Qualification

Question Qualification evaluates each candidate question together with its completion pair. It retains questions that are relevant, remain unresolved by S, and pass behavioral verification. The qualified set is

$$Q_{\text{qual}} = \{q \in Q_{\text{gen}} \mid \text{Rel}(q) \wedge \text{Unres}(q) \wedge \text{Diff}(q)\}.$$

Relevance and Resolution. Relevance checks whether q concerns required program behavior and whether its completions specify different alternatives for that behavior. Resolution then checks whether S already determines the answer. A question is marked as resolved only when text in S supports one completion and rules out the other. The judgment must cite this supporting text. Only relevant and unresolved questions proceed to behavioral verification.

Behavioral Verification. For each completion h_z, where $z \in \{A, B\}$, CONTRA independently generates three programs p_z^1, p_z^2, p_z^3 conditioned on S, q, and h_z. It also generates a shared set $\mathcal{X}$ of at least seven inputs valid under both completions using S, q, h_A, and h_B.

All six programs are executed on the shared inputs. For each completion, the output on input x is considered stable when at least two programs agree,

$$o_z(x) = y \quad \Longleftrightarrow \quad \left|\left\{j \in \{1,2,3\} \mid p_z^j(x) = y\right\}\right| \geq 2.$$

If no output receives at least two votes, $o_z(x)$ is undefined. The question passes behavioral verification when the two groups produce different stable outputs on at least one shared input,

$$\begin{aligned} \text{Diff}(q) = 1 \quad \Longleftrightarrow \quad & \exists x \in \mathcal{X} \text{ such that} \\ & o_A(x), o_B(x) \text{ are defined and } o_A(x) \neq o_B(x). \end{aligned}$$

Otherwise, $\text{Diff}(q) = 0$, and the question is discarded.

This comparison tests the consequences of alternative answers to a specific question. Shared inputs make the two groups directly comparable, while agreement within each group reduces the influence of an isolated implementation error. A question is retained only when execution provides evidence of a behavioral difference. A failed check does not establish behavioral equivalence.

3.5 Adaptive Question Selection

A user response may resolve another qualified question or make it unnecessary. Therefore, Adaptive Question Selection uses the interaction history to reassess which questions still require user input.

Let $Q_t \subseteq Q_{\text{qual}}$ denote the remaining questions at step t, with $Q_1 = Q_{\text{qual}}$. Given S, H_t, and the text of the questions in Q_t, the selector either chooses a question that still requires user input or returns STOP when further clarification is unnecessary,

$$\text{Select}(S, H_t, Q_t) \in Q_t \cup \{\text{STOP}\}.$$

After receiving a_t, CONTRA appends (q_t, a_t) to the history and removes q_t from the candidate set. The updated history helps avoid questions that have become resolved or redundant. Clarification ends when the selector returns STOP, no questions remain, or B questions have been asked.

4 Experiments

4.1 Experimental Setup

Benchmark and Evaluation. We evaluate on ClarifyCodeBench (Fang et al., 2026), which provides underspecified programming requirements with gold clarification questions and answers. Our evaluation uses 419 underspecified tasks and the 80 fully specified tasks provided by ClarifyCodeBench.

Table 1: Clarification results (%) on ClarifyCodeBench. P and R denote precision and recall. Bold values indicate the best result for each metric and coding agent.

GPT-5-mini				
Method	F1 ↑	TKQR ↑	P ↑	R ↑
Direct Prompting	16.86	15.09	20.62	14.25
Ask-or-Assume	17.06	17.00	17.06	17.06
ClarifyGPT	19.92	30.94	13.58	**37.37**
CONTRA	**35.36**	**34.32**	**36.20**	34.56

Grok 4.5				
Method	F1 ↑	TKQR ↑	P ↑	R ↑
Direct Prompting	23.12	17.24	**41.81**	15.98
Ask-or-Assume	29.17	25.54	37.25	23.97
ClarifyGPT	20.90	27.83	16.16	29.59
CONTRA	**36.54**	**35.95**	36.58	**36.50**

Gemini 3.1 Pro				
Method	F1 ↑	TKQR ↑	P ↑	R ↑
Direct Prompting	37.12	33.93	44.68	31.75
Ask-or-Assume	37.55	35.54	43.47	33.05
ClarifyGPT	17.14	18.64	16.60	17.71
CONTRA	**50.49**	**45.37**	**58.74**	**44.28**

GPT-5.5				
Method	F1 ↑	TKQR ↑	P ↑	R ↑
Direct Prompting	24.10	16.29	**59.32**	15.12
Ask-or-Assume	25.51	21.64	34.31	20.30
ClarifyGPT	23.53	26.53	20.03	28.51
CONTRA	**42.39**	**37.63**	50.15	**36.72**

Each method may ask at most five clarification questions before generating a final program. The benchmark's GPT-4o evaluator matches questions against each task's gold questions by majority vote over three judgments.

Baselines. We compare CONTRA with three training-free baselines. Direct Prompting uses the benchmark prompt to ask a question or generate code. Ask-or-Assume (Edwards & Schuster, 2026) adds an intent agent that reviews the proposed action and requests clarification when needed. ClarifyGPT (Mu et al., 2024) generates questions from behavioral disagreement among programs sampled from the current requirement and executed on shared inputs. We also compare with the coding harnesses Claude Code (Anthropic, 2025) and OpenHands (Wang et al., 2025).

Evaluation Metrics. We report question precision, recall, F1, and turn-discounted key question rate (TKQR) (Fang et al., 2026). Precision is the proportion of all questions asked that match a previously uncovered gold question, including questions asked on fully specified tasks in the denominator. Recall is the proportion of gold questions matched by at least one question, and F1 is the harmonic mean of precision and recall. TKQR rewards earlier identification of gold questions using normalized discounted cumulative gain (Järvelin & Kekäläinen, 2002), averaged across tasks. For the harness comparison, we also report the false-positive rate (FPR): the proportion of fully specified tasks on which at least one question is asked.

Models and Implementation. We evaluate coding agents built on GPT-5-mini, GPT-5.5, Grok 4.5, and Gemini 3.1 Pro. For each comparison, all components of CONTRA and the baselines use the same LLM. We use low reasoning effort, with stochastic sampling only for program generation in Behavioral Verification and ClarifyGPT. The harness comparison uses Qwen3.8-27B for CONTRA, Claude Code, and OpenHands, with the same question budget and scoring procedure. We execute generated Python programs in isolated environments under fixed time, memory, and output limits.

4.2 MAIN RESULTS

CONTRA achieves the highest F1 and TKQR for every coding agent (Table 1). Its macro-average F1 and TKQR reach 41.20% and 38.32%, exceeding the best baselines by 13.88 and 12.33 percentage points, respectively. Table 2 further shows that CONTRA achieves the highest macro-average precision and recall. These gains reflect more key questions identified and a lower proportion of unnecessary questions.

Table 2: Macro-average results (%) across the four agents. Bold values indicate the best result.

Method	F1 ↑	TKQR ↑	P ↑	R ↑
Direct Prompting	25.30	20.64	41.61	19.28
Ask-or-Assume	27.32	24.93	33.02	23.60
ClarifyGPT	20.37	25.99	16.59	28.29
CONTRA	**41.20**	**38.32**	**45.42**	**38.01**

The baseline results illustrate the difficulty of identifying key questions while avoiding unnecessary ones. With GPT-5.5, Direct Prompting achieves 59.32% precision but only 15.12% recall, leaving many key questions unasked. CONTRA increases recall to 36.72% while maintaining 50.15% precision. Conversely, with GPT-5-mini, ClarifyGPT achieves 37.37% recall but only 13.58% precision, indicating that many questions do not identify a previously uncovered gold question. Here, CONTRA raises precision to 36.20% while retaining a comparable recall of 34.56%. Despite slightly lower recall, CONTRA improves TKQR from 30.94% to 34.32%, indicating that it finds key questions earlier and uses clarification turns more efficiently.

The relative performance of baseline methods also varies across coding agents. For example, ClarifyGPT achieves the highest baseline recall with GPT-5-mini but the lowest with Gemini 3.1 Pro. This variation suggests that its effectiveness depends strongly on the coding agent. By comparison, CONTRA consistently achieves the highest F1 and TKQR across all four agents, showing that the gains hold across the evaluated coding agents.

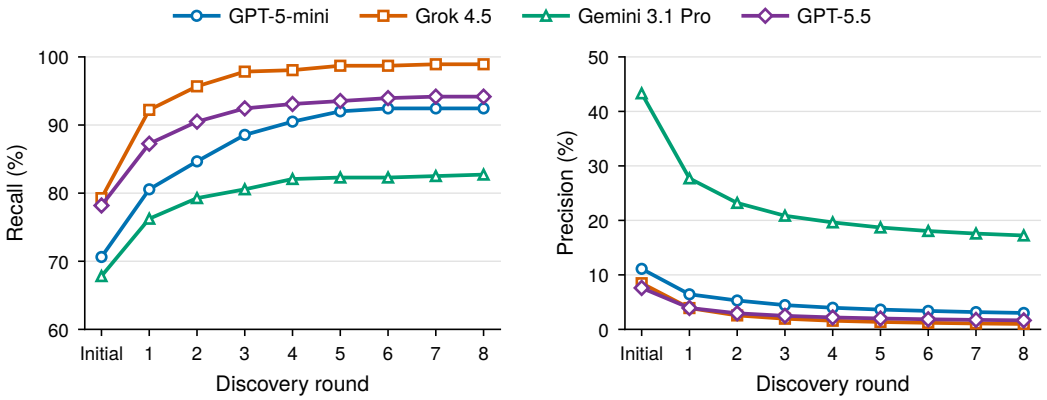

Figure 2: Recall and precision across discovery rounds. Each point evaluates all distinct candidates found up to that round. Initial denotes the first generation step.

4.3 EFFECT OF DISCOVERY ROUNDS

Question Discovery achieves high recall across all four coding agents, recovering most gold clarification questions within approximately four discovery rounds. Figure 2 shows how recall and precision change as candidate questions accumulate. Recall rises rapidly in early rounds before saturating, indicating that iterative discovery finds key questions missed initially.

However, precision declines as the candidate set grows. This indicates that additional rounds also introduce many questions that do not identify additional gold questions. Therefore, broad discovery provides a candidate set with high recall, while Question Qualification filters unnecessary questions from this set.

4.4 EFFECT OF QUESTION QUALIFICATION

Figure 3 shows that F1 increases at every qualification stage across all four coding agents. Relevance and Resolution improve candidate quality, and Behavioral Verification provides further gains. These consistent improvements demonstrate the effectiveness of successive filtering in balancing the retention of key questions with the removal of unnecessary ones.

Filtering alone also improves TKQR. Across the four coding agents, macro-average TKQR increases from 19.03% after Question Discovery to 32.52% after Behavioral Verification. The qualification checks only remove candidates and preserve the relative order of the remaining questions. Removing unnecessary questions allows key questions to appear earlier in the sequence. Beyond improving candidate quality, successive filtering thus brings key questions forward, making more effective use of the question budget.

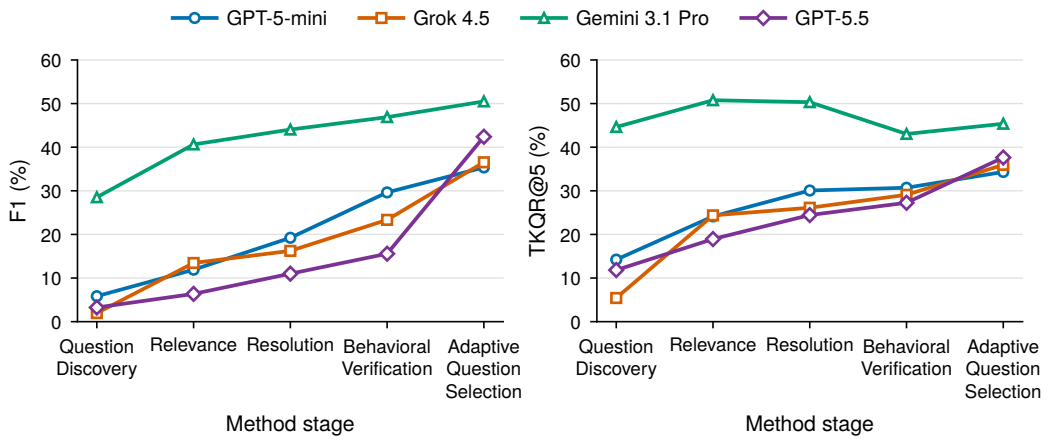

Figure 3: F1 and TKQR@5 across CONTRA stages. Before Adaptive Question Selection, F1 evaluates all retained candidates and TKQR@5 evaluates the first five questions in discovery order.

4.5 EFFECT OF ADAPTIVE QUESTION SELECTION

Adaptive Question Selection uses the interaction history to avoid asking qualified questions that user answers have already resolved or made unnecessary.

The final transition in Figure 3 shows that Adaptive Question Selection further improves both F1 and TKQR across all four coding agents. Macro-average precision increases from 21.57% to 45.42%, while recall decreases from 50.16% for the full qualified candidate set to 38.01% for the questions actually asked. This transition compares the full candidate set with a five-question interaction. The resulting macro-average F1 increases from 28.86% to 41.20%, and TKQR rises from 32.52% to 38.32%. These gains show more efficient identification of key questions within the five-question budget.

4.6 ABLATION ANALYSIS

Table 3 shows that removing any component reduces F1 for GPT-5-mini. The full method achieves the highest precision and F1.

Relevance checks required behavior, while Resolution checks whether the requirement already supplies an answer. Removing Relevance lowers precision from 36.20% to 31.27%, while removing Resolution lowers it to 26.46%. The larger decline highlights the need to avoid asking for information already supplied in the requirement.

Table 3: Component ablations (%) with GPT-5-mini. Bold marks the best result for each metric.

Variant	F1 ↑	TKQR ↑	P ↑	R ↑
CONTRA	**35.36**	34.32	**36.20**	34.56
w/o REL	32.54	32.91	31.27	33.91
w/o RES	29.13	32.98	26.46	32.40
w/o BV	32.58	**37.10**	28.04	**38.88**
w/o AQS	26.69	30.70	20.96	36.72

REL, Relevance; RES, Resolution; BV, Behavioral Verification; AQS, Adaptive Question Selection.

Behavioral Verification further improves precision. Removing it increases recall and TKQR, but lowers precision from 36.20% to 28.04% and F1 from 35.36% to 32.58%. This trade-off supports checking how candidate answers affect program behavior before retaining questions.

Removing Adaptive Question Selection causes the largest drop in precision and F1. F1 falls by 8.67 percentage points, while TKQR decreases from 34.32% to 30.70%, despite higher recall. These results highlight the value of interaction history in selecting useful questions within a limited budget.

Removing Behavioral Verification and removing Adaptive Question Selection both increase recall, but only the former improves TKQR. Therefore, finding more key questions does not by itself ensure that they are asked early. This contrast highlights their different effects on question timing.

5 PRACTICAL EVALUATION AND DEPLOYMENT

5.1 EFFECT ON DOWNSTREAM CODE GENERATION

Each agent generates a final program from the original underspecified requirement and the complete clarification history. We evaluate these programs using the benchmark test cases.

Table 4 reports the highest pass@1 for CONTRA with both agents on underspecified tasks. With Gemini 3.1 Pro, it reaches 57.32%, exceeding Direct Prompting by 1.22 percentage points. With GPT-5.5, it reaches 64.63%, exceeding ClarifyGPT by 0.61 percentage points. These modest gains across both agents suggest that effective clarification can improve program correctness.

Table 4: Downstream pass@1 (%) on ClarifyCodeBench. Bold marks the best result.

Method	Gemini 3.1 Pro	GPT-5.5
Direct Prompting	56.10	62.50
Ask-or-Assume	55.49	63.72
ClarifyGPT	53.35	64.02
CONTRA	**57.32**	**64.63**

Clarification quality measures whether questions recover the intended requirements; pass@1 additionally measures whether the agent implements those requirements correctly. Reporting both captures these two stages of the interaction.

5.2 COMPARISON WITH CODING HARNESSES

We compare CONTRA with Claude Code and OpenHands on ClarifyCodeBench. All three use Qwen3.8-27B under the same question budget and evaluation protocol.

Table 5 shows that CONTRA achieves the highest F1 and TKQR, reaching 19.86% and 18.79%, respectively. These exceed the best harness results by 9.25 and 9.80 percentage points. Its recall reaches 20.73%, compared with 6.26% for Claude Code and 8.42% for OpenHands, indicating that more key clarification questions are identified under the same budget.

Table 5: Clarification results (%) with Qwen3.8-27B. Bold marks the best result for each metric.

Method	F1↑	TKQR↑	P↑	R↑	FPR↓
Claude Code	9.63	6.59	**20.86**	6.26	**5.00**
OpenHands	10.61	8.99	14.34	8.42	32.50
CONTRA	**19.86**	**18.79**	19.05	**20.73**	**5.00**

FPR is measured on fully specified tasks.

The recall gain over Claude Code comes with a small precision decrease, from 20.86% to 19.05%. Its 5.00% FPR matches Claude Code and is below OpenHands at 32.50%. Therefore, CONTRA identifies more key questions on underspecified tasks without increasing the frequency of unnecessary interaction on fully specified tasks relative to Claude Code.

5.3 EFFICIENT CLARIFICATION AND PLUGIN DEPLOYMENT

On-Demand Question Qualification. To reduce the cost of Question Qualification, we introduce CONTRA-Flash, which qualifies candidates on demand. Adaptive Question Selection first chooses a candidate from the discovered questions. The candidate then undergoes Relevance, Resolution, and Behavioral Verification before it can be asked. Rejected candidates are discarded and selection continues. This ordering avoids qualifying candidates that the selector never considers.

Efficiency and Clarification Quality. Figure 4 shows that CONTRA-Flash reduces candidate-level qualification evaluations by 65.7–96.0%. F1 remains stable for three of the four coding agents: GPT-5-mini, Grok 4.5, and GPT-5.5 reduce qualification evaluations by 85.6%, 83.3%, and 96.0%, respectively, while F1 changes by at most 1.8 percentage points. Flash exceeds the best baseline F1 for all three agents and the best baseline TKQR for Grok 4.5 and GPT-5.5. For Gemini 3.1 Pro, Flash achieves 35.7% F1 and 28.3% TKQR with 65.7% fewer qualification evaluations, placing its F1 within 1.9 percentage points of the best baseline. Overall, Flash offers substantial computational savings with stable F1 on most evaluated agents.

Integration with Claude Code. We implement CONTRA as a Claude Code plugin that reviews decisions after implementation. The plugin combines the agent's own report with an independent

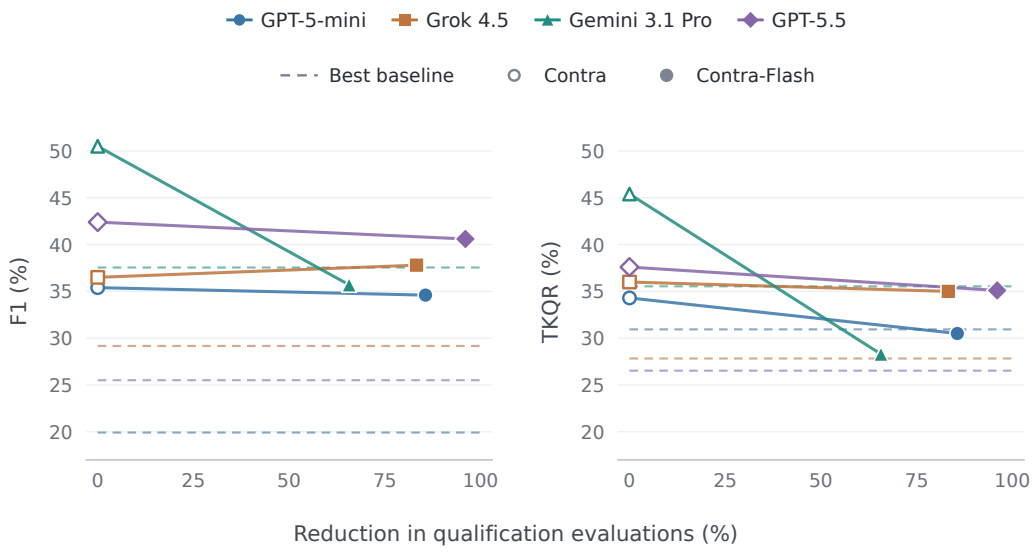

Figure 4: Efficiency and quality of CONTRA-Flash. Open/filled markers denote CONTRA/Flash. Dashed lines show the best baseline for each agent and metric in Table 1.

review of the requirement and code changes to discover unresolved decisions. It then prioritizes these decisions and explains what the requirement leaves unspecified, what the agent implemented, and a plausible alternative. For unresolved implementation choices, Behavioral Verification runs in the background and uses distinguishing tests to check whether alternative implementations produce different observable behaviors. Plugin implementation details appear in Appendix B.1.

Repository Evaluation. We compare Claude Code with and without the plugin on 40 single-function tasks from commits in six Python repositories. Independent review identifies 116 of 262 reported decisions (44.3%) absent from the agent's own report. We also assess whether the agent's reported decisions explain unspecified information, the implemented choice, and an alternative interpretation.

Table 6: Completeness of reported decisions (%).

Information reported	Claude Code	+CONTRA
Unspecified information	4.3	59.7
Alternative interpretation	7.1	47.8
All three elements	2.9	**46.3**

Claude Code and +CONTRA report 70 and 67 decisions, respectively.

Table 6 shows increases from 4.3% to 59.7% for unspecified information and from 7.1% to 47.8% for alternative interpretations. Reports covering all three elements also rise from 2.9% to 46.3%. Therefore, the plugin surfaces unreported decisions and makes unresolved choices explicit. Independent review expands the decisions available for inspection, while the structured report explains why each choice may require confirmation. Together, they help developers assess whether an implemented choice matches their intent. Evaluation details appear in Appendix B.2.

6 CONCLUSION

We presented CONTRA, a training-free method for selective clarification in coding agents. It combines Question Discovery, Question Qualification, and Adaptive Question Selection. Semantic checks and execution evidence identify consequential choices, while interaction history guides which questions still require user input. Experiments show consistent improvements in F1 and TKQR across four coding agents and gains over existing coding harnesses. CONTRA-Flash reduces qualification evaluations, while the Claude Code plugin makes unresolved implementation choices and their alternatives explicit for developer review. Future work will evaluate how selective clarification affects developer effort and code quality over longer development sessions.

AI USE STATEMENT

We used generative AI tools to draft, translate, and edit manuscript text, check references, assist with figure design, and generate the illustrative cartoon in Appendix A. LLM use for program and test generation, clarification evaluation, repository task labeling, and report assessment is described in Sections 3–4 and Appendix B; all 40 repository task labels received manual review. The authors take full responsibility for the final content.

REPRODUCIBILITY STATEMENT

Section 3 describes the three stages of CONTRA, including program sampling and the behavioral verification criteria. Section 4.1 specifies the benchmark, baselines, models, question budget, and evaluation protocol, while Section 5.1 describes downstream code evaluation. Appendix B details the plugin implementation and repository evaluation, including software and model versions, task sampling and its random seed, manual label review, and report-quality assessment.

REFERENCES

Anthropic. Claude Code, 2025. URL https://github.com/anthropics/claude-code. Software repository. Accessed September 26, 2026.

Nicholas Edwards and Sebastian Schuster. Ask or assume? uncertainty-aware clarification-seeking in coding agents. *arXiv preprint arXiv:2603.26233*, 2026. URL https://arxiv.org/abs/2603.26233.

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. doi: 10.1109/TSE.2024.3428972.

Zheng Fang, Dongming Jin, Yihong Dong, Yongmin Li, Kechi Zhang, Zhi Jin, and Ge Li. ClarifyCodeBench: Evaluating LLMs on clarifying ambiguous requirements for code generation. *arXiv preprint arXiv:2607.00711*, 2026. URL https://arxiv.org/abs/2607.00711.

Google DeepMind. Gemini 3.1 Pro model card, 2026. URL https://deepmind.google/models/model-cards/gemini-3-1-pro/.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations*, volume 2024, pp. 23247–23275, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

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

Haoxiang Jia, Robbie Morris, He Ye, Federica Sarro, and Sergey Mechtaev. Automated repair of ambiguous problem descriptions for LLM-based code generation. *arXiv preprint arXiv:2505.07270*, 2025. URL https://arxiv.org/abs/2505.07270.

Lorenz Kuhn, Yarin Gal, and Sebastian Farquhar. CLAM: Selective clarification for ambiguous questions with generative language models. *arXiv preprint arXiv:2212.07769*, 2022. URL https://arxiv.org/abs/2212.07769.

Haau-Sing (Xiaocheng) Li, Mohsen Mesgar, André Martins, and Iryna Gurevych. Python code generation by asking clarification questions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 14287–14306, 2023. doi: 10.18653/v1/2023.acl-long.799. URL https://aclanthology.org/2023.acl-long.799/.

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. doi: 10.1145/3660810.

OpenAI. GPT-5 system card, 2025. URL https://openai.com/index/gpt-5-system-card/.

OpenAI. GPT-5.5 system card, 2026. URL https://openai.com/index/gpt-5-5-system-card/.

Qwen Team. Qwen3.8-27B, 2026. URL https://huggingface.co/Qwen/Qwen3.8-27B. Model card. Accessed September 26, 2026.

SpaceXAI. Introducing Grok 4.5, 2026. URL https://x.ai/news/grok-4-5.

Zhao Tian and Junjie Chen. Aligning requirement for large language model's code generation. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, 2026. doi: 10.1145/3744916.3764572. URL https://doi.org/10.1145/3744916.3764572.

Sanidhya Vijayvargiya, Vijay Viswanathan, and Graham Neubig. Asking what matters: Reward-driven clarification for software engineering tasks. *arXiv preprint arXiv:2604.14624*, 2026a. URL https://arxiv.org/abs/2604.14624.

Sanidhya Vijayvargiya, Xuhui Zhou, Akhila Yerukola, Maarten Sap, and Graham Neubig. AmbigSWE: Interactive agents to overcome underspecificity in software engineering. In *International Conference on Learning Representations*, 2026b. URL https://openreview.net/forum?id=X2yzXtH4wp.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

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. URL https://arxiv.org/abs/2504.16331.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652, 2024. doi: 10.52202/079017-1601.

Michael Zhang, W. Bradley Knox, and Eunsol Choi. Modeling future conversation turns to teach LLMs to ask clarifying questions. In *International Conference on Learning Representations*, volume 2025, pp. 60722–60742, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/97e2df4bb8b2f1913657344a693166a2-Abstract-Conference.html.

Michael JQ Zhang and Eunsol Choi. Clarify when necessary: Resolving ambiguity through interaction with LMs. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 5541–5558, 2025. doi: 10.18653/v1/2025.findings-naacl.306. URL https://aclanthology.org/2025.findings-naacl.306/.

A ONE REQUEST, FIVE DECISIONS: AN ILLUSTRATIVE WALKTHROUGH

Our plugin demonstration follows a short request in Requests: *"Add Brotli decoding support."* The recorded change and its five reported decisions show what CONTRA exposes. Figure 5 extends this example with a hypothetical later user complaint to illustrate why these decisions matter.



Figure 5: A completion report can leave implementation choices implicit. Those choices may become consequential later. The plugin makes them available for review while the requirement and change are still in context.

A reassuring completion report. The agent replaces the fixed `Accept-Encoding` value `gzip, deflate` with `DEFAULT_ACCEPT_ENCODING`. Its final reply says that Brotli is included when available and that the Quickstart documentation has been updated. To the developer, the request appears complete. Yet the report does not explain how the new behavior interacts with dependency versions, which encodings the header may advertise, or whether the new constant is intended to become public API.

The choice that surfaces later. In the demonstration, Brotli support requires `urllib3` 1.25 or later, while the package still declares a minimum of 1.21.1. Imagine a user returning three weeks later with version 1.24: the header falls back to the older encodings, although the documentation describes Brotli support. The request never specified whether to raise the dependency bound, warn about unavailable support, or accept the fallback. Each policy has different compatibility implications. The delayed complaint illustrates why a seemingly local edit can leave a consequential decision unresolved.

Five decisions become reviewable. With the plugin, the same change is followed by the structured report in Figure 6. Two entries concern changes beyond the explicit request: introducing a public name in `requests.utils` and updating the Quickstart documentation. Two concern deferred work: the dependency-bound mismatch and missing regression coverage for the new header. The fifth identifies an implementation choice: forwarding all encodings detected by `urllib3`, including `zstd`, instead of adding only `br`. Together, these entries make scope, deferred work, and alternative interpretations available for review.

From an observation to executable evidence. For the fifth entry, the demonstration links a distinguishing test, `tests/test_contra_default_headers.py`, showing a behavioral difference between the implemented encoding policy and an alternative. Appendix B.1 describes the verification procedure. The developer can inspect this evidence, confirm the intended behavior, and retain the test as a record. The resulting discussion is concrete: should this feature enable Brotli alone, or delegate the advertised encoding set to the dependency?

B Plugin Implementation and Evaluation Details

B.1 Plugin Implementation

The Claude Code plugin reviews implementation decisions at the end of an interaction. A hook records the working tree when the user submits a request. When the agent finishes, another hook compares the updated tree with this snapshot to identify changes made during the interaction. This avoids including unrelated changes that were already present before the request.

The plugin then runs two processes in parallel. One extracts decisions from the agent's final report using `claude-haiku-4-5-20251001`. The other independently reviews the requirement, code changes, and repository context using `claude-opus-5`, without reading that report. Their outputs are merged using text overlap for entries concerning the same file and decision category. The combined report distinguishes implementation choices, changes beyond the requested scope, and work the agent decided not to perform. Rule-based prioritization uses these categories together with affected scope, trigger conditions, reversibility, and verification outcomes.

For an unresolved implementation choice, the plugin uses `claude-opus-5` to generate a distinguishing test and a patch implementing an alternative interpretation. It executes the test on the current and patched implementations in isolated repository copies. A test that passes on the current implementation and fails on an executable alternative provides evidence of a behavioral difference. Patch or test-execution errors are distinguished from such evidence. In normal use, verification runs in the background after the initial report. Distinguishing tests can be retained in the repository to document the implemented behavior, and a persistent record tracks decisions across interactions.

B.2 Repository Evaluation

The evaluation uses 40 single-function commits from six Python repositories, namely Rich, Flask, Requests, Click, HTTPX, and Typer. The tasks are selected by stratified random sampling, with 20 labeled underspecified and 20 labeled fully specified, using seed 20260903. For each task, the repository is restored to its pre-commit state, and the commit message serves as the requirement. Claude Code is asked to modify the target function and briefly explain the change. The final plugin runs use Claude Code 2.1.261 with `claude-opus-5` and the research version of the plugin evaluated on September 5, 2026. Each task has one baseline run and one plugin run. Verification runs synchronously during evaluation so that its outcomes are collected before the process exits.

Task labels are generated by `claude-opus-5` from the commit message and pre-change code, then manually reviewed for all 40 tasks. The criterion is whether this information determines the required observable behavior, without filling missing information from implementation conventions.

Report quality is evaluated from the agent's final reply. We use `claude-opus-5` to extract decision entries and check whether each states the unspecified information, the implemented choice, and a plausible alternative interpretation. Table 6 reports the proportion of entries containing each element and all three together. The denominators are 70 entries without the plugin and 67 with it.

The decision-discovery analysis uses the plugin's combined structured output, a separate collection from the final-reply entries above. Among its 262 decisions, 116 appear only in independent review, 89 only in the agent-derived list, and 57 in both. The repository evaluation thus measures decision discovery and report completeness.

C Plugin Interface and Operational Scope

Figure 6 presents an English rendering of the demonstration. The request, code change, and completion report remain visible above the five decisions, connecting each concern to the interaction that produced it.

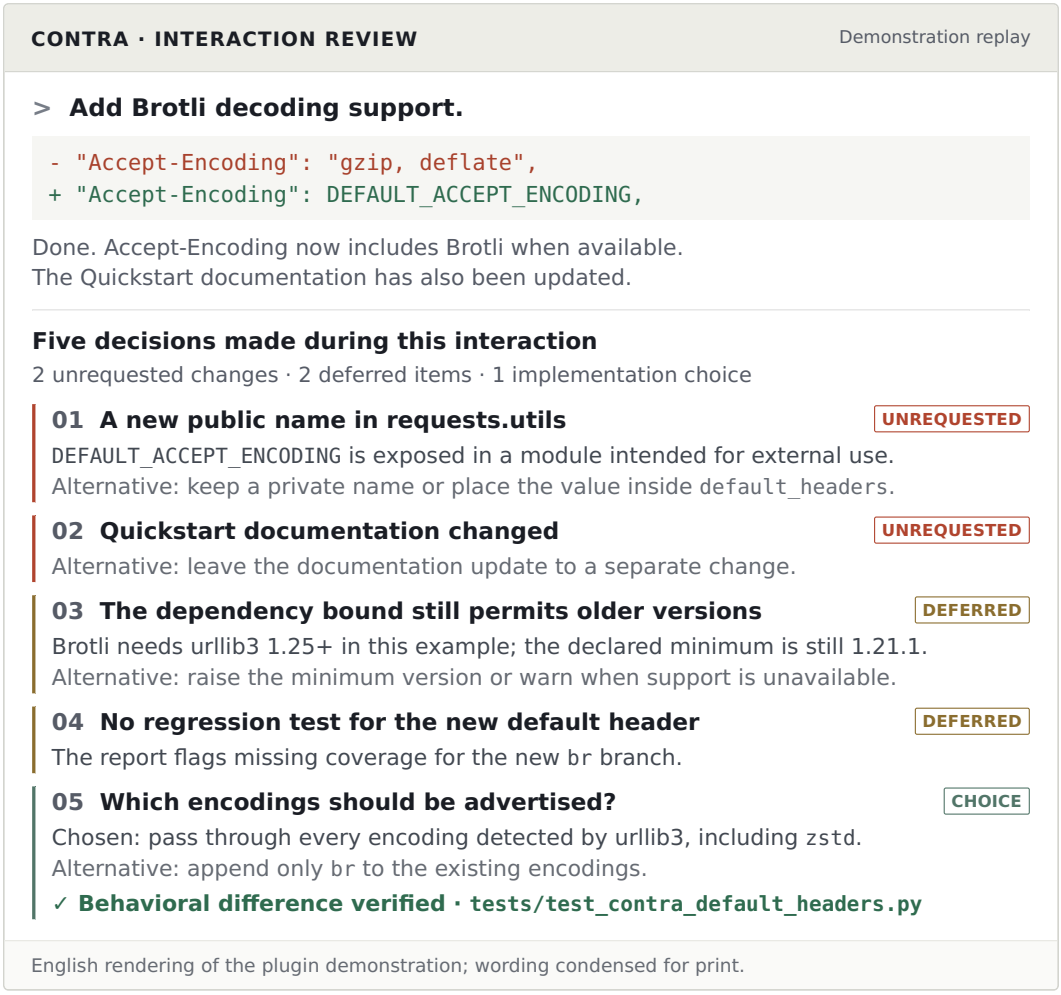

Figure 6: English rendering of the supplied plugin demonstration, condensed for print. It preserves the five decisions, three categories, and test-artifact path. The verification indicator applies to the fifth entry.

Reading the report. An entry exposes the implemented choice or deferred work and, where available, an alternative. The developer can accept the choice, request a revision, or defer it explicitly. A separate verification indicator identifies entries supported by an observed behavioral difference.

Cost and applicability. This single-run demonstration reports 40 s ($0.47) for coding, 90 s ($0.17) for reporting, and 15 s ($0.06) for background verification. The report appears about 130 s after the request; the total reported cost, including verification, is $0.70. In this build, executable evidence uses Python and `pytest`; other supported source languages receive decision reports. Verification supports the behavior exercised by each distinguishing test.